A Dynamic Approach to Defuse Logic Bombs in Android Applications

Fausto Fasano$^{1,2[0000-0003-3736-6383]}$, Michele Guerra$^{1,2,3[0000-0001-5507-9084]}$, Roberto Milanese^{1}, and Rocco Oliveto$^{1,3[0000-0002-7995-8582]}$

[1] University of Molise, Department of Bioscience and Territory, Pesche (IS), Italy
{`fausto.fasano, michele.guerra, roberto.milanese, rocco.oliveto`}`@unimol.it`
[2] MOSAIC Research Center, DiBT, Unimol
[3] Stake Lab, DiBT, Unimol

Abstract. Logic bombs are a critical security threat in Android applications that can be triggered by specific events or conditions, leading to serious consequences. In this work we focus on a special type of logic bomb exploiting mobile device resources for sensitive data leakage. Such malicious behaviour can exploit Android permission model by gaining access to sensitive related resources in a legitimate context and later using them in a dangerous one, once the logic bomb is triggered. To address this scenario, we propose a dynamic approach to avoid privacy leakage in case the logic bomb is triggered. To this aim, we extended RPCDroid, a tool that monitors the behavior of an Android application whenever it accesses specific device resources. In particular, to defuse the logic bomb we force an explicit prompt to authorize dangerous requests, leveraging the execution context to prevent accesses unbeknownst to the user, still minimizing the effort required. We assessed the effectiveness of our proposal using TriggerZoo, a publicly available dataset containing apps with injected logic bombs. Our results show that a context aware enabled permission model can effectively prevent uncontrolled access to privacy related data in case a logic bomb is triggered.

Keywords: Security and Privacy · Context Aware Permission Model · Dynamic Analysis · Android Permission Model

1 Introduction

Android is the most popular mobile operating system[4], with over 2.5 billion active users worldwide. Given its pervasive use in a wide range of user devices, whether handheld, at home, or in the office, Android's security and privacy have become paramount concerns. With this growing popularity, the security of Android applications has become a significant concern, as new threats are discovered regularly, even in the official Google Play app store[15]. One of the most significant threats to the security of Android applications is logic bombs.

[4] https://www.idc.com/promo/smartphone-market-share

A logic bomb is a type of malicious code that is triggered by specific events or conditions. When activated, a logic bomb can execute various harmful actions, such as stealing sensitive data or crashing the system.

Thousands of apps are regularly flagged by antimalware engines. Actually, the AndroZoo [2] repository has collected over 22 millions of malicious app versions, among which over 19 millions were available on Google Play Store. Therefore, addressing the spread of malware in app markets is a prime concern for researchers and practitioners.

Unfortunately, automatically detecting logic bomb is still an open issue, due to several pitfalls, such as the high rate of false positives since benign and malicious apps can use the same code for benign and/or malicious behavior, thus requiring an analyst to verify the behavior [26]. To address this issue, there is a need for a more context-aware approach to permission control in Android applications. Context awareness refers to an application's ability to adapt its behavior based on the current use context.

In recent years, there has been growing interest in using dynamic analysis techniques for detecting logic bombs in Android applications. This is mainly due to the fact that static analysis can be deceived by different obfuscation schemes [14]. Dynamic analysis refers to the analysis of an application's behavior during runtime. Dynamic analysis techniques have been used in various security contexts, including detecting malware and identifying vulnerabilities in web applications [3][29][32].

However, approaches resilient to various bomb analysis techniques including fuzzing, symbolic execution, multi-path exploration, and program slicing have been proposed [36].

In this work, we adopt a complementary approach in which, admitting the possibility that an app contains a logic bomb, we monitor the resources accessed by the app and regulate their access based on the usage context. Thereby, we are not affected by the false positive problem. It is worth noting that our approach does not aim to provide a solution to the problem of logic bombs, but we address a scenario in which an untrusted app as well as a malware that deceived a bomb analysis tool is executed. In particular, we do not aim at identifying the presence of the logic bomb, but we aim at preventing it from causing access to protected resources unbeknownst to the user. With the term protected resources we refer to all those features of the device that are associated with potentially sensitive data leakage and which Android therefore protects by requiring access to such a dangerous permission. In case the app has not been granted such permission, it cannot access the resource. Examples of protected resources are: camera, microphone, contacts, SMS, location, and many more.

Although Android has improved the permission management system over the years, it still suffers from some limitations that allow a logic bomb to access, under certain conditions, these resources without the user's awareness. One of the mechanisms that can be easily implemented by a malicious app to bypass Android's protection is to hide the malicious behavior within apps which, due to the functionality they offer, legitimately have access to one or more of these protected resources. Once access permission is obtained, the app triggers the logic bomb without the operating system deeming it necessary to request for further confirmation.

Clearly, logic bombs are designed to trigger this behavior only if the user has previously permanently granted access to the resource they want to exploit, so as not to betray their presence by trying to access unauthorized resources. Android allows the user to specify whether access to a resource should be limited to a single execution or whether it is persistent. However, it is very likely that if the legit functionality is frequent – sooner or later – the user will decide to grant the permission permanently. In such a circumstance the logic bomb would be triggered.

To mitigate this problem we propose an extension of the Android permissions model in which the authorization to access a protected resource is associated to the execution context. By execution context we mean a combination of the functionality the app is involved in and the user interface element the user interacted with. There are several ways to identify the execution context. For instance, it is possible to analyze the calling context, i.e., the list of active functions currently on the call stack [30], use the calling context encoding [31], the execution index [33], as well as techniques to recording and reporting dynamic calling contexts [4]. By adopting the calling context as a new dimension of the permission model, the user will be able to continue using the functionality s/he has already granted permissions to without the burden of repeatedly allowing access to the same resource, but at the same time any access to the same protected resource occurring in contexts different from already authorized ones will require an explicit authorization. As a result, the logic bomb will not trigger if the context has not been previously analyzed. On the other hand, if the logic bomb was designed to trigger independently of the context, e.g., because it is unaware of the modified permission model, access to protected resources would have to be explicitly granted by the user. This behavior would be an alarm bell for the user, who could identify it as malicious, thus defusing or mitigating the bomb effect.

In this paper, to identify the calling contexts and instrument the context-based permission model, we extend RPCDroid [12], a tool for monitoring the resources accessed by Android applications. RPCDroid is designed to monitor any access to protected resources at runtime by tracking the invocation of sensitive APIs and system services and logging useful information to determine the context a protected resource is used in. Note that these are only a subset of the possible execution context, namely those involving protected resources usage. A logic bomb not using such type of resources could be triggered without being intercepted. As said before, in this study we simply aim at avoiding such kind of malicious behaviour. Clearly, the approach could be extended to include more sophisticated context identification approaches and different type of resources currently not protected by dangerous permissions.

We use RPCDroid both to enforce the permission model with context-aware access control policies and to better understand logic bomb common behaviours to further improve the tool ability to detect anomalies in the accesses to protected resources. By analyzing the logs generated by RPCDroid, we can indeed identify the exact conditions under which a logic bomb is triggered, providing a better understanding of its behavior and how it is activated.

To assessed the effectiveness of our proposal to identify when a logic bomb is triggered and control the access to a protected resources, we used TriggerZoo [27], a publicly available dataset containing apps with injected logic bombs. We manually executed a subset of the apps and, using the indications provided in the dataset, we managed to invoke the app features that contain the triggers for the logic bombs. The achieved results confirm that using a context-aware permission model leveraging dynamic analysis techniques, we can prevent sensitive data from being leaked without the knowledge of the user even in case of a logic bomb.

The rest of the paper is organized as follows. Section 2 describes the context aware permission model implemented with RPCDroid. In Section 3 we provide a preliminary evaluation of the effectiveness of our proposal using TriggerZoo. Section 4 discussed related works, while Section 5 concludes the paper and gives indications for future work.

2 Context Aware Permission Model with RPCDroid

In this paper, we aim to develop a comprehensive approach to assess the effectiveness of adopting the usage context to discriminate between different types of access to the same sensitive resource in Android applications. Our primary goal is to enhance user security and privacy by modifying the permission model to be context aware, prompting users to make a choice whenever permissions are used in a new context. This approach is based on the premise that the logic bomb triggering event is closely related to the misuse of permissions. Our system relies on user interaction to detect and mitigate such threats.

Our approach offers several advantages over existing approaches. It allows us to identify context-specific behavior, reducing the risk of false positives and enabling us to focus our efforts on the most critical contexts. Furthermore, in addition to identifying the specific context in which a logic bomb is triggered, we propose an approach to defuse it. Specifically, our approach focuses on the use of permissions, which Android uses to grant applications access to sensitive resources such as the camera, microphone, and contacts. Our approach requires explicit user consent before an application can use a protected resource in a specific context, thereby reducing the risk of malicious applications misusing permissions to trigger logic bombs. To implement our approach, we improve the existing Android permission model, which requires developers to declare the permissions that their applications require and prompt the user for consent before using them. We extend this model by incorporating context awareness and fine-grained permission control, which allow us to limit the use of permissions to specific contexts and provide users with finer-grained control over the permissions granted to applications. Our approach is based on the premise that a logic bomb is triggered by a specific sequence of events, which can be identified by analyzing the runtime behavior of an application.

2.1 RPCDroid

We used and improved upon an existing dynamic analysis tool called RPCDroid to monitor the execution of mobile applications that access specific device resources requiring dangerous permissions. Our enhancements focus on creating a finer-grained approach that allows for a better understanding of the logic bomb triggering process and helps prevent malicious behavior activation in Android applications. Indeed, RPCDroid was designed to perform a dynamic analysis of the use of dangerous permissions in Android. The output of the tool consists of

- One or more `JSON` files containing a log of all events handled (such as an Activity change or interaction with a UI component)
- One or more `JSON` files files containing the permissions used
- Additional material useful for the analysis of the identified contexts, such as *Screenshots* and *screen recording* for each event or permission request

RPCDroid Analyzer is installed as an EXposed module on the Android device or emulator and collects information to identify resource usage contexts. Upon starting the monitored application, RPCDroid Analyzer tracks any access to resources that require dangerous permissions at the system level (e.g., camera, microphone, storage, or location) through a dynamic injection mechanism based on hooking and callback techniques. These include all the actions performed by the user at the interface level and any system call to the Android access control and validation mechanism. More details on the tool are available in [12].

2.2 Enabling Permission Control in RPCDroid

RPCDroid has been modified to save each context in which the request for a protected resource has previously been analysed. To prevent some logic bombs from not triggering due to the denial of the corresponding permissions, we decided to automatically grant the permissions declared by the app within the Manifest file, in order to put tool in the worst possible scenario. In fact, one of the most common behaviors for logic bombs is to not trigger if they don't have the opportunity to execute the malicious payload. By granting these permissions outside of the trigger code, we simulated a situation where the logic bomb is free to trigger at any time. Moreover, we improved the existing Android permission model, which requires developers to declare the permissions that their applications require and prompt the user for consent before using them. We extended this model by using context information provided by RPCDroid, allowing the user to grant or deny a previously evaluated permission request anytime the execution context is different (see Figure 1) providing users with finer-grained control over the permissions granted to applications.

Currently, the system identifies a context based on user actions on the UI (such as touching a button or changing activities) and permissions requested at

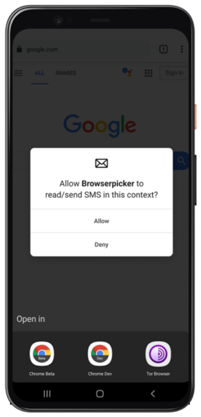
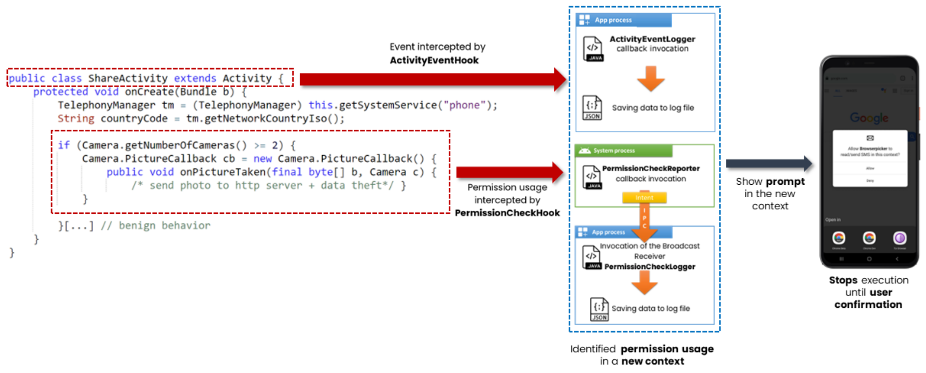


Fig. 1: Example of added *prompt*

runtime. It is worth noting that the approach can be enhanced to include more restrictive or exhaustive execution context identification approaches. The pro-



```
public class ShareActivity extends Activity {
    protected void onCreate(Bundle b) {
        TelephonyManager tm = (TelephonyManager) this.getSystemService("phone");
        String countryCode = tm.getNetworkCountryIso();

        if (Camera.getNumberOfCameras() >= 2) {
            Camera.PictureCallback cb = new Camera.PictureCallback() {
                public void onPictureTaken(final byte[] b, Camera c) {
                    /* send photo to http server + data theft*/ }
            }
        }[...] // benign behavior
    }
}
```

Event intercepted by **ActivityEventHook**
Permission usage intercepted by **PermissionCheckHook**
App process
ActivityEventLogger callback invocation
Saving data to log file
System process
PermissionCheckReporter callback invocation
Intent
IPC
App process
Invocation of the Broadcast Receiver **PermissionCheckLogger**
Saving data to log file
Identified **permission usage** in a **new context**
Show **prompt** in the new context
Stops execution until **user confirmation**

Fig. 2: The approach *dynamically identifies* individual permission usage contexts, showing the user the *decision prompt*.

posed approach is depicted in Figure (Figure 2). RPCDroid uses the EXposed framework to dynamically inject the services we developed. These services perform method hooking concerning the invocation of dangerous permissions and use callbacks to inject code that dynamically displays the permission management prompt.

Finally, we monitor the app's behavior in the application process by intercepting the calls to dangerous permissions associated to protected resources. We use this information to contextualize the permission requests and display the appropriate prompt to the user. In this way, we can provide a more fine-grained permission usage system that stops the execution of applications until the user chooses whether allowing the access or not.

3 Evaluation

To evaluate the effectiveness of our context aware permission model in preventing a logic bomb to actually access sensitive data without the knowledge of the user, we executed the improved RPCDroid tool on a dataset of Android apps called *TriggerZoo* [27], provided by Androzoo [2]. This dataset contains 406 apps with **injected logic bombs** and includes descriptions of the logic bomb activation (e.g., the activity in which it is activated, permissions used, and return values). We executed our tool on an Android 11 (API 30) emulator with RPCDroid active. We randomly selected a subset including 70 apps, executed them using our modified version of the RPCDroid monitoring tool, and conducted an *in-depth analysis of their behavior* to understand the logic bomb triggering process and its relationship with permission misuse.

In Table 1 we report the results of our study. In particular, 45 of the selected apps could not be executed due to various reasons, while for 25 of them we were able to reproduce all the steps needed to test the trigger condition.

Category	**Number of Apps**
Total apps analyzed	70
Non-executable apps	45
Dataset related issues	36
Incorrect permission declaration	10
Incompatible with Android version	6
Other errors	20
RPCDroid related issues	9
Executable apps	25
RPCDroid mitigation successful	16
Trigger condition not met	4
App closed by logic bomb	3
RPCDroid mitigation failed	2

Table 1: Summary of the analysis results for the 70 apps

Amongst the reasons that prevented the app from being executed without crashing we mention *issues in the repackaged application*, *incorrect permission declarations within the Manifest*, and *incompatibility with the emulator's Android versions* (i.e., Android 11). Specifically, 36 apps were not executable due to dataset related issues such as corrupted APKs. We verified this issue by attempting to execute the apps on different emulators with various Android versions. We ruled out the possibility that our tool was the cause of the application crash, executing them on different emulators where the tool was not installed and experiencing the same issues. Among these 36 apps, 10 had errors related to the incorrect injection of permission declaration within the Manifest by TriggerZoo. For example, in the XML reported in Listing 1.1 instead of declaring the permission as `android.permission.ACCESS_FINE_LOCATION` it has been declared as `Manifest.permission.ACCESS_FINE_LOCATION`. In our emulator, this caused the app crashing as soon as the permission was requested, as Android did not recognize it.

Listing 1.1: Example of incorrect permission declaration in the manifest

```
<manifest
 xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.prueba.joel.sort">
 <uses-permission
        android:name="Manifest.permission.
            ACCESS_FINE_LOCATION"/>
<uses-permission
        android:name="Manifest.permission.
            ACCESS_COARSE_LOCATION"/>
<uses-permission
        android:name="android.permission.INTERNET"/>
</manifest>
```

Furthermore, 6 of the 36 apps were incompatible with the Android version used in our analysis due to the outdated target SDK used to build them. The remaining 20 apps had issues with method calls and native code or were improperly recompiled, causing them to crash upon startup. The TriggerZoo authors acknowledged that such issues might be possible. Finally, 9 of the apps failed to execute when RPCDroid was active. These were hybrid apps that, for some reason, attempted to use permissions not declared in the Manifest. Since RPCDroid sets all manifest permissions to "Granted" during the analysis, using undeclared permission caused our tool to throw an exception and terminate the app execution. It is important to note that such undeclared permissions are no more allowed in recent versions of Android, so this issue would not exist in a real setting.

Considering the 25 apps that correclty executed on the emulator, we *manually executed* them while our enhanced RPCDroid tool monitored the running session. In 16 of them, the tool displayed a **prompt asking us to allow or deny using a sensitive resource** precisely when the trigger event that would activate the logic bomb occurred. As a result, the app execution was paused until we decided on permission use in that specific context. It is worth noting that the context in which the tool requested resource usage was *off-topic* compared to the requested resource. This is consistent with what we expected from malicious behavior. For example, a request happened in a context where the app displayed a list of images to set as the device's wallpaper. When we pressed the button to set the wallpaper, the app requested permission to send/read SMS. In this situation, we naturally denied the usage, preventing the trigger event activation. To further evaluate whether that was indeed a trigger event, we retraced our steps through the same context while confirming the permission use this time. Through the log analysis, we were able to identify that the event was, in fact, the trigger for a logic bomb. It is important to emphasize that such behavior occurred consistently across all the analyzed apps.

In addition to the 16 apps mentioned earlier, we analyzed four other applications in which the trigger condition for the logic bomb could not be activated due to the design of the apps. Specifically, one of these apps required authentication through a login form whose information was not provided in the dataset description. Two apps were designed to activate the logic bomb only if the device met certain requirements, such as specific IMEI or model information which, unfortunately, are limited on the emulator. Lastly, one of the apps is meant to activate the logic bomb at a specific time of a particular day. By analyzing the behavior of these apps as described by the authors of Triggerzoo, we are confident that, had we met the trigger condition, our RPCDroid enforcement would have correctly presented the user with a prompt to block or allow the use of the protected resources requested by the logic bomb.

For three applications, once the trigger condition for the logic bomb was satisfied, the malicious behavior of the bomb involved closing the application, but our tool did not display any prompt. Although in these cases the code behind the trigger was actually executed without the user being able to prevent it, it should be noted that our approach is specifically designed to control access to protected device resources, so we expected that this behavior would not have been intercepted by the tool as it is not associated with a permission in Android. Similarly, for two applications the tool was unable to mitigate the behavior of the logic bomb since the malicious behavior did not involve using dangerous permissions.

It is worth noting that when running the apps manually, requests for permissions were almost exclusively made during the activation of a logic bomb. This is probably also due to the limited complexity of the apps in the dataset which do not include many features. However, the mechanism for detecting the execution context and monitoring only dangerous permissions has proven effective in limiting repeated requests to the end user. Moreover, from the analysis of the RPCDroid logs it is also possible to extract common patterns in the use of permissions by applications containing logic bombs. This could further improve the tool's ability to intercept the most dangerous requests in a more precise manner, potentially providing suggestions to direct the user in the decision to grant or deny a permission request.

To conclude, in our analysis of 25 apps, the proposed RPCDroid enhancement was effective in 16 apps, where the prompt was shown to users, allowing them to deny the access to protected resources requested by the logic bomb. The trigger condition could not be activated in four apps, so we could not assess the approach against them. In the rest of the apps the tool could not mitigate the logic bomb behavior due to the lack of permission usage during the malicious action, but no privacy violations could occurr in similar situations. This is a remarkable result concerning the overall effectiveness of the context aware fine grained permission model to defuse logic bombs or mitigate their impact on user privacy. The system's ability to associate UI actions and permission requests to specific contexts allowed for a more comprehensive understanding of the application's behavior and the potential risks involved.

4 Related Works

Over the past decade, numerous approaches have been proposed to automate malware detection. These approaches involve exploring static analysis techniques [9,10,13,19,37], dynamic execution [21], or a combination of both [5,6,34], as well as machine learning [20,25]. Although effective on benchmarks, these techniques may fail to detect new attacks that use sophisticated evasion techniques. For example, attackers may employ code obfuscation [7] to bypass static analysis or hide malicious code behind triggering conditions during dynamic analysis.

Logic bombs can be used for various malicious activities, such as adware [8], Trojan [22], ransomware [35], spyware [24], and more [40]. As the trigger and the malicious code are independent of the core application code, logic bombs can easily be added to legitimate apps and repackaged for distribution [11,16,17,39]. Therefore, detecting logic bombs is crucial, particularly in mobile devices containing critical personal information. Despite the challenges in detecting logic bombs, various approaches have been proposed in the literature. For instance, researchers have explored static-analysis-based heuristic or machine learning approaches [18] and dynamic-analysis-based approaches [10,21,38] to identify suspicious sensitive triggers. However, detecting logic bombs remains a problem [23], with both static and dynamic analyses often unable to detect such behaviors [1] [36].

Static analyses may be limited by many conditional statements in a given app code, making it challenging to identify suspicious sensitive triggers accurately. Moreover, high rate of false positives suggest that a manual analysis should be conducted to prevent a legitimate app from being erroneously considered to contain a logic bomb [26].

TriggerScope [10] uses symbolic execution to detect logic bombs but is limited to certain trigger types and may not scale to large datasets. Dark Hazard [18] leverages a supervised learning approach with engineered features to identify sensitive triggers but flags up to 20% of apps, making it inefficient for isolating dangerous triggers. Unlike these approaches, our proposed method focuses specifically on identifying suspicious unexpected dangerous events and detecting logic bombs among them using dynamic analysis.

Difuzer [28] proposed an approach that relies on unsupervised learning techniques and specific trigger/behavior features to identify suspicious hidden sensitive operations. In contrast, our approach focuses on dynamic analysis and context-aware detection to prevent logic bombs triggered in dangerous contexts from being executed without the user approval, allowing for targeted mitigation strategies. Additionally, our approach does not block app execution in case of a false positive but instead informs the user of sensitive resource usage, allowing for continued app use. In the event of a false positive, our system does not block the execution of the app or prevent the user from acting. Instead, it informs the user that a sensitive resource will be used in that context, allowing the user to make an informed decision about whether to proceed or not. This capability is an improvement over existing techniques that rely solely on binary classification and often generate false positives, leading to unnecessary blocking of app execution.

5 Conclusion

The growing popularity of Android and the increasing number of threats affecting mobile apps demand innovative approaches to ensure user security and privacy.

In this work, we addressed logic bombs, a type of malicious code that is triggered by specific events or conditions. We proposed a dynamic approach to avoid privacy leakage in case the malicious code is activated. In particular, we aim at identifying execution contexts related to protected resources and prevent unauthorized access to privacy related data through logic bombs. Unlike other related approaches, we do not specifically focus on identifying logic bombs, but provide a mechanism to mitigate the effects of potential malware by controlling access to protected resources. This prevent us from suffering from the false positives issue. Indeed, our approach does not block the execution of the app in the case of a candidate logic bomb. Instead, we informs users whenever a sensitive resource is going to be used in a specific context, enabling users to block the malicious behavior.

We assessed the effectiveness of our proposal with TriggerZoo [27], a publicly available dataset containing apps with injected logic bombs. The results are promising since the approach allowed us to prevent the execution of malicious code behind a logic bomb trigger in all the situations in which the code actually tried to access a protected resource. We also collected logs generated by our enhanced version of RPCDroid, a tool that monitors the runtime behavior of Android applications to identify the exact conditions under which a logic bomb is triggered and permit a better understanding of its behavior. This gave us important insights on the behaviour of logical bombs while accessing privacy related resources that we plan to use to further improve the ability to distinguish legitimate and malicious behaviors, reducing the effort required to the user for decisions regarding dangerous permissions.

The tool used to assess the proposed approach simply represents a prototype implementation. In fact, it requires root access to the mobile device making it unsuitable for a real context. However, it is desirable that in the future, a permission management aware of the execution context and the usage pattern of dangerous permissions will be natively integrated into the operating system.

In this way, the approach could actually contributes to the overall security of the Android ecosystem and protects user privacy and sensitive data even in case of security threats like logic bombs, notoriously difficult to detect.

References

1. Agrawal, H., Alberi, J.L., Bahler, L., Micallef, J., Virodov, A., Magenheimer, M., Snyder, S., Debroy, V., Wong, E.: Detecting hidden logic bombs in critical infrastructure software (2012)
2. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories. pp. 468–471. MSR '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2901739.2903508, `http://doi.acm.org/10.1145/2901739.2903508`
3. Bellizzi, J., Vella, M.: Wexpose: Towards on-line dynamic analysis of web attack payloads using just-in-time binary modification. 2015 12th International Joint Conference on e-Business and Telecommunications (ICETE) **04**, 5–15 (2015)
4. Bond, M.D., Baker, G.Z., Guyer, S.Z.: Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In: ACM-SIGPLAN Symposium on Programming Language Design and Implementation (2010)
5. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D.X., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Botnet Detection (2008)
6. Choudhary, M., Kishore, B.: Haamd: Hybrid analysis for android malware detection. In: 2018 International Conference on Computer Communication and Informatics (ICCCI). pp. 1–4 (2018). https://doi.org/10.1109/ICCCI.2018.8441295
7. Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., Xu, F., Chen, K., Wang, X., Zhang, K.: Understanding android obfuscation techniques: A large-scale investigation in the wild. In: Beyah, R., Chang, B., Li, Y., Zhu, S. (eds.) Security and Privacy in Communication Networks. pp. 172–192. Springer International Publishing, Cham (2018)
8. Erturk, E.: A case study in open source software security and privacy: Android adware. In: World Congress on Internet Security (WorldCIS-2012). pp. 189–191 (2012)
9. Fereidooni, H., Conti, M., Yao, D., Sperduti, A.: Anastasia: Android malware detection using static analysis of applications. In: 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5 (2016). https://doi.org/10.1109/NTMS.2016.7792435
10. Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G.: Triggerscope: Towards detecting logic bombs in android applications. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 377–396 (2016). https://doi.org/10.1109/SP.2016.30
11. Gadyatskaya, O., Lezza, A.L., Zhauniarovich, Y.: Evaluation of resource-based app repackaging detection in android. pp. 135–151 (11 2016). https://doi.org/10.1007/978-3-319-47560-8$_9$
12. Guerra., M., Milanese., R., Oliveto., R., Fasano., F.: Rpcdroid: Runtime identification of permission usage contexts in android applications. In: Proceedings of the 9th International Conference on Information Systems Security and Privacy - ICISSP,. pp. 714–721. INSTICC, SciTePress (2023). https://doi.org/10.5220/0011797200003405

13. Kang, H., Jang, J.w., Mohaisen, D., Kim, H.K.: Detecting and classifying android malware using static analysis along with creator information. International Journal of Distributed Sensor Networks **2015**, 1–9 (06 2015). https://doi.org/10.1155/2015/479174
14. Khalid, S., Hussain, F.B.: Evaluating dynamic analysis features for android malware categorization. In: 2022 International Wireless Communications and Mobile Computing (IWCMC). pp. 401–406 (2022). https://doi.org/10.1109/IWCMC55113.2022.9824225
15. Kotzias, P., Caballero, J., Bilge, L.: How did that get in my phone? unwanted app distribution on android devices. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 53–69 (2021). https://doi.org/10.1109/SP40001.2021.00041
16. Li, L., Bissyande, T.F., Klein, J.: Rebooting research on detecting repackaged android apps: Literature review and benchmark. IEEE Transactions on Software Engineering **47**(04), 676–693 (apr 2021). https://doi.org/10.1109/TSE.2019.2901679
17. Li, L., Bissyandé, T.F., Klein, J.: Simidroid: Identifying and explaining similarities in android apps. In: 2017 IEEE Trustcom/BigDataSE/ICESS. pp. 136–143 (2017). https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230
18. Pan, X., Wang, X., Duan, Y., Wang, X., Yin, H.: Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps (01 2017). https://doi.org/10.14722/ndss.2017.23265
19. Papp, D., Buttyán, L., Ma, Z.: Towards semi-automated detection of trigger-based behavior for software security assurance. In: Proceedings of the 12th International Conference on Availability, Reliability and Security. ARES '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3098954.3105821, https://doi.org/10.1145/3098954.3105821
20. Peiravian, N., Zhu, X.: Machine learning for android malware detection using permission and api calls. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. pp. 300–305 (2013). https://doi.org/10.1109/ICTAI.2013.53
21. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of android malware. Proceedings of the 7th European Workshop on System Security, EuroSec 2014 (04 2014). https://doi.org/10.1145/2592791.2592796
22. Pieterse, H., Olivier, M.S.: Android botnets on the rise: Trends and characteristics. In: 2012 Information Security for South Africa. pp. 1–5 (2012). https://doi.org/10.1109/ISSA.2012.6320432
23. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**, 358–366 (1953)
24. Saad, M.H., Serageldin, A., Salama, G.I.: Android spyware disease and medication. In: 2015 Second International Conference on Information Security and Cyber Forensics (InfoSec). pp. 118–125 (2015). https://doi.org/10.1109/InfoSec.2015.7435516
25. Sahs, J., Khan, L.: A machine learning approach to android malware detection. In: 2012 European Intelligence and Security Informatics Conference. pp. 141–147 (2012). https://doi.org/10.1109/EISIC.2012.34
26. Samhi, J., Bartel, A.: On the (in)effectiveness of static logic bomb detection for android apps. IEEE Transactions on Dependable and Secure Computing **19**(6), 3822–3836 (2022). https://doi.org/10.1109/TDSC.2021.3108057
27. Samhi, J., Bissyandé, T.F., Klein, J.: Triggerzoo: A dataset of android applications automatically infected with logic bombs. In: Proceedings of the

19th International Conference on Mining Software Repositories. p. 459–463. MSR '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524842.3528020, `https://doi.org/10.1145/3524842.3528020`

28. Samhi, J., Li, L., Bissyand'e, T.F., Klein, J.: Difuzer: Uncovering suspicious hidden sensitive operations in android apps. 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE) pp. 723–735 (2021)
29. Sekar, R.C.: An efficient black-box technique for defeating web application attacks. In: Network and Distributed System Security Symposium (2009)
30. Sumner, W.N., Zhang, X.: Identifying execution points for dynamic analyses. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 81–91 (2013). https://doi.org/10.1109/ASE.2013.6693069
31. Sumner, W.N., Zheng, Y., Weeratunge, D., Zhang, X.: Precise calling context encoding. IEEE Transactions on Software Engineering **38**(5), 1160–1177 (2012). https://doi.org/10.1109/TSE.2011.70
32. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krügel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: Network and Distributed System Security Symposium (2007)
33. Xin, B., Sumner, W.N., Zhang, X.: Efficient program execution indexing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 238–248. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375611, `https://doi.org/10.1145/1375581.1375611`
34. Xu, L., Zhang, D., Jayasena, N., Cavazos, J.: Hadm: Hybrid analysis for detection of malware. pp. 702–724 (09 2018). https://doi.org/10.1007/978-3-319-56991-$8_5$1
35. Yang, T., Yang, Y., Qian, K., Lo, D.C.T., Qian, Y., Tao, L.: Automated detection and analysis for android ransomware. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems. pp. 1338–1343 (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.39
36. Zeng, Q., Luo, L., Qian, Z., Du, X., Li, Z., Huang, C.T., Farkas, C.: Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs. IEEE Transactions on Dependable and Secure Computing **18**(6), 2582–2600 (2021). https://doi.org/10.1109/TDSC.2019.2957787
37. Zhao, Q., Zuo, C., Dolan-Gavitt, B., Pellegrino, G., Lin, Z.: Automatic uncovering of hidden behaviors from input validation in mobile apps. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 1106–1120 (2020). https://doi.org/10.1109/SP40000.2020.00072
38. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. p. 93–104. SPSM '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2381934.2381950, `https://doi.org/10.1145/2381934.2381950`
39. Zhou, W., Zhang, X., Jiang, X.: Appink: Watermarking android apps for repackaging deterrence. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. p. 1–12. ASIA CCS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2484313.2484315, `https://doi.org/10.1145/2484313.2484315`

40. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy. pp. 95–109 (2012). https://doi.org/10.1109/SP.2012.16