

Chapter 2

Background

Abstract Android applications are developed on top of Android framework and therefore bear particular features compared to traditional desktop software. In the meantime, due to the unique design and implementation, Android apps are threatened by emerging cyber attacks that target at mobile operating systems. As a result, security researchers have made considerable efforts to discover, mitigate and defeat these threats.

2.1 Android Application

Android is a popular operating system for mobile devices. It dominates in the battle to be the top smartphone system in the world, and ranked as the top smartphone platform with 52 % market share (71.1 million subscribers) in Q1 2013. The success of Android is also reflected from the popularity of its applications. Tens of thousands of Android apps become available in Google Play while popular apps (e.g., Adobe Flash Player 11) have been downloaded and installed over 100 million times.

Android apps are developed using mostly Java programming language, with the support of Android Software Development Kit (SDK). The source code is first compiled into Java classes and further compiled into a Dalvik executable (i.e., DEX file) via `dx` tool. Then, the DEX program and other resource files (e.g., XML layout files, images) are assembled into the same package, called an APK file. The APK package is later submitted to the Android app markets (e.g., Google Play Store) with the developer's descriptions in text and other formats. An app market serves as the hub to distribute the application products, while consumers can browse the market and purchase the APK files. Once a APK file is downloaded and installed to a mobile device, the Dalvik executable will be running within a Dalvik virtual machine (DVM).

2.1.1 Android Framework API

While an APK file is running in DVM, the Android framework code is also loaded and executed in the same domain. As a matter of fact, a Dalvik executable merely acts as a plug-in to the framework code, and a large portion of program execution happens within the Android framework.

A DEX file interacts with the Android framework via Application Programming Interface (API). These APIs are provided to the developers through Android SDK. From developers' perspective, Android API is the only channel for them to communicate with the underlying system and enable critical functionalities. Due to the nature of mobile operating system, Android offers a broad spectrum of APIs that are specific to smartphone capabilities. For instance, an Android app can programmatically send SMS messages via `sendTextMessage()` API or retrieve user's geographic location through `getLastKnownLocation()`.

2.1.2 Android Permission

Sensitive APIs are protected by Android permissions. Android exercises an install-time permission system. To enable the critical functionalities in an app, a developer has to specify the needs for corresponding permissions in a manifest file `AndroidManifest.xml`. Once an end user agrees to install the app, the required permissions are granted. At runtime, permission checks are enforced at both framework and system levels to ensure that an app has adequate privileges to make critical API calls.

There exist two major limitations for this permission system. Firstly, once certain permission is granted to an app at the install time, there is no easy way to revoke it at runtime. Secondly and more importantly, the permission enforcement is fundamentally a single-point check and thus lacking continuous protection. If an application can pass a checkpoint and retrieve sensitive data via a critical API call, it can use the data without any further restrictions.

2.1.3 Android Component

Android framework also provides a special set of APIs that are associated to Android components. Components in Android are the basic building units for apps. In particular, there exist four types of components in Android: Activity, Service, Broadcast Receiver, and Content Provider. An Activity class takes care of creating the graphical user interface (GUI) and directly interacts with the end user. A Service, in contrast, performs non-interactive longer-running operations in background while accepting service requests from other apps or app

components. A Broadcast Receiver is component that listens to and processes system-wide broadcast messages. A Content Provider encapsulates data content and shares it with multiple components via a unified interface.

Components communicate with one another via *Intents*. An Intent with certain ACTION code, target component and payload data indicates a specific operation to be performed. For example, with different target parameter, an Intent can be used to launch an Activity, request a Service or send a message to any interested Broadcast Receiver. A developer can create custom permissions to protect components from receiving Intents from an arbitrary sender. However, such a simple mechanism cannot rule out malicious Intent communication because it does not prevent a malicious app author from requesting the same custom permission at install time.

2.1.4 Android App Description

Once an Android app has been developed, it is delivered to the app markets along with the developer's descriptions. Developers are usually interested in describing the app's functionalities, unique features, special offers, use of contact information, etc. Nevertheless, they are not motivated to explain the security risks behind the sensitive app functions. Prior studies [33, 36] have revealed significant inconsistencies between what the app is claimed to do and what the app actually does. This indicates that a majority of apps exercise undeclared sensitive functionalities beyond the users' expectation. Such a practice may not necessarily be malicious, but it does provide a potential window for attacks to exploit.

To mitigate this problem, Android markets also explains to users, in natural language, what permissions are required by an app. The goal is to help users understand the program behaviors so as to avoid security risks. However, such a simple explanation is still too technical for average users to comprehend. Besides, a permission list does not illustrate how permissions are used by an app. As an example, if an application first retrieves user's phone number and then sends it to a remote server, it in fact uses two permissions, READ_PHONE_STATE and INTERNET in a collaborative manner. Unfortunately, the permission list can merely inform that two independent permissions have been used.

2.2 Android Malware Detection

The number of new Android malware instances has grown exponentially in recent years. McAfee reports [28] that 2.47 million new mobile malware samples were collected in 2013, which represents a 197 % increase over 2012. Greater and greater

amounts of manual effort are required to analyze the increasing number of new malware instances. This has led to a strong interest in developing methods to automate the malware analysis process.

Existing automated Android malware detection and classification methods fall into two general categories: (1) signature-based and (2) machine learning-based. Signature-based approaches [17, 54] look for specific patterns in the bytecode and API calls, but they are easily evaded by bytecode-level transformation attacks [37]. Machine learning-based approaches [1, 2, 34] extract features from an application's behavior (such as permission requests and critical API calls) and apply standard machine learning algorithms to perform binary classification. Because the extracted features are associated with application syntax, rather than program semantics, these detectors are also susceptible to evasion.

2.2.1 Signature Detection and Malware Analysis

Previous studies were focused on large-scale and light-weight detection of malicious or dangerous Android apps. DroidRanger [54] proposed permission-based footprinting and heuristics-based schemes to detect new samples of known malware families and identify certain behaviors of unknown malicious families, respectively. Risk-Ranker [17] developed an automated system to uncover dangerous app behaviors, such as root exploits, and assess potential security risks. Kirin [11] proposed a security service to certify apps based upon predefined security specifications. WHYPER [33] leveraged Natural Language Processing and automated risk assessment of mobile apps by revealing discrepancies between application descriptions and their true functionalities. Efforts were also made to pursue in-depth analysis of malware and application behaviors. TaintDroid [12], DroidScope [47] and VetDroid [51] conducted dynamic taint analysis to detect suspicious behaviors during runtime. Ded [13], CHEX [25], PEG [6], and FlowDroid [3] exercised static dataflow analysis to identify dangerous code in Android apps. The effectiveness of these approaches depends upon the quality of human crafted detection patterns specific to certain dangerous or vulnerable behaviors.

2.2.2 Android Malware Classification

Many efforts have also been made to automatically classify Android malware via machine learning. Peng et al. [34] proposed a permission-based classification approach and introduced probabilistic generative models for ranking risks for Android apps. Juxtapp [18] performed feature hashing on the opcode sequence to detect malicious code reuse. DroidAPIMiner [1] extracted Android malware features at the API level and provided light-weight classifiers to defend against malware installations. DREBIN [2] took a hybrid approach and considered both

Android permissions and sensitive APIs as malware features. To this end, it performed broad static analysis to extract feature sets from both manifest files and bytecode programs. It further embedded all feature sets into a joint vector space. As a result, the features contributing to malware detection can be analyzed geometrically and used to explain the detection results. Despite the effectiveness and computational efficiency, these machine learning based approaches extract features from solely external symptoms and do not seek an accurate and complete interpretation of app behaviors.

2.3 Android Application Vulnerabilities

Although the permission-based sandboxing mechanism enforced in Android can effectively confine each app's behaviors by only allowing the ones granted with corresponding permissions, a vulnerable app with certain critical permissions can perform security-sensitive behaviors on behalf of a malicious app. It is so called confused deputy attack. This kind of security vulnerabilities can present in numerous forms, such as privilege escalation [8], capability leaks [16], permission re-delegation [14], content leaks and pollution [53], component hijacking [25], etc.

Prior work primarily focused on automatic discovery of these vulnerabilities. Once a vulnerability is discovered, it can be reported to the developer and a patch is expected. Some patches can be as simple as placing a permission validation at the entry point of an exposed interface (to defeat privilege escalation [8] and permission re-delegation [14] attacks), or withholding the public access to the internal data repositories (to defend against content leaks and pollution [53]), the fixes to the other problems may not be so straightforward.

2.3.1 Component Hijacking Vulnerabilities

In particular, component hijacking may fall into the latter category. When receiving a manipulated input from a malicious Android app, an app with a component hijacking vulnerability may exfiltrate sensitive information or tamper with the sensitive data in a critical data repository on behalf of the malicious app. In other words, a dangerous information flow may happen in either an outbound or inbound direction depending on certain external conditions and/or the internal program state.

A prior effort has been made to perform static analysis to discover *potential* component hijacking vulnerabilities [25]. Static analysis is known to be conservative in nature and may raise false positives. For instance, static analysis may find a viable execution path for information flow, which may never be reached in actual program execution; static analysis may find that interesting information is stored in some elements in a database, and thus has to conservatively treat the entire database

as sensitive. As a result, upon receiving a discovered vulnerability, the developer has to manually confirm if the reported vulnerability is real. However, it is nontrivial for average developers to properly fix the vulnerability and release a patch.

2.3.2 Automatic Patch and Signature Generation

While an automated patching method is still lacking for vulnerable Android apps, a series of studies have been made to automatically generate patch for conventional client-server programs. AutoPaG [23] analyzes the program source code and identifies the root cause for out-of-bound exploit, and thus creates a fine-grained source code patch to temporarily fix it without any human intervention. IntPatch [50] utilizes classic type theory and dataflow analysis framework to identify potential integer-overflow-to-buffer-overflow vulnerabilities, and then instruments programs with runtime checks. Sidiroglou and Keromytis [39] rely on source code transformations to quickly apply automatically created patches to vulnerable segments of the targeted applications, that are subject to zero-day worms. Newsome et al. [31] propose an execution-based filter which filters out attacks on a specific vulnerability based on the vulnerable program's execution trace. ShieldGen [7] generates a data patch or a vulnerability signature for an unknown vulnerability, given a zero-day attack instance. Razmov and Simon [38] automate the filter generation process based on a simple formal description of a broad class of assumptions about the inputs to an application.

2.3.3 Bytecode Rewriting

In principle, these aforementioned patching techniques can be leveraged to address the vulnerabilities in Android apps. Nevertheless, to fix an Android app, a specific bytecode rewriting technique is needed to insert patch code into the vulnerable programs. Previous studies have utilized this technique to address varieties of problems. The Privacy Blocker application [35] performs static analysis of application binaries to identify and selectively replace requests for sensitive data with hard-coded shadow data. I-ARM-Droid [9] rewrites Dalvik bytecode to interpose on all the API invocations and enforce the desired security policies. Aurasium [46] repackages Android apps to sandbox important native APIs so as to monitor security and privacy violations. Livshits and Jung [24] implement a graph-theoretic algorithm to place mediation prompts into bytecode program and thus protect resource access. However, due the simplicity of the target problems, prior work did not attempt to rewrite the bytecode program in an extensive fashion. In contrast, to address sophisticated vulnerabilities, such as component hijacking, a new machinery has to be developed, so that inserted patch code can effectively monitor and control sensitive information flow in apps.

2.3.4 Instrumentation Code Optimization

The size of a rewritten program usually increases significantly. Thus, an optimization phase is needed. Several prior studies attempted to reduce code instrumentation overhead by performing various static analysis and optimizations. To find error patterns in Java source code, Martin et al. optimized dynamic instrumentation by performing static pointer alias analysis [27]. To detect numerous software attacks, Xu et al. inserted runtime checks to enforce various security policies in C source code, and remove redundant checks via compiler optimizations [45]. As a comparison, due to the limited resources on mobile devices, there exists an even more strict restriction for app size. Therefore, a novel method is necessary to address this new challenge.

2.4 Privacy Leakage in Android Apps

While powerful Android APIs facilitate versatile functionalities, they also arouse privacy concerns. Previous studies [12, 13, 19, 44, 52, 54] have exposed that both benign and malicious apps are stealthily leaking users' private information to remote servers. Efforts have also been made to detect and analyze privacy leakage either statically or dynamically [12, 13, 15, 22, 25, 26, 48]. Nevertheless, a good solution to defeat privacy leakage at runtime is still lacking.

2.4.1 Privacy Leakage Detection

Egele et al. [10] studied the privacy threats in iOS applications. They proposed PiOS, a static analysis tool to detect privacy leaks in Mach-O binaries. TaintDroid is a dynamic analysis tool for detecting and analyzing privacy leaks in Android applications [12]. It modifies Dalvik virtual machine and dynamically instruments Dalvik bytecode instructions to perform dynamic taint analysis. Enck et al. [13] proposed a static analysis approach to study privacy leakage as well. They convert a Dalvik executable to Java source code and leverage a commercial Java source code analysis tool Fortify360 [20] to detect surreptitious data flows. CHEX [25] is designed to vet Android apps for component hijacking vulnerabilities and is essentially capable of detecting privacy leakage. It converted Dalvik bytecode to WALA [43] SSA IR, and conducted static dataflow analysis with WALA framework. AndroidLeaks [15] is a static analysis framework, which also leverages WALA, and identifies potential leaks of personal information in Android applications on a large scale. Mann et al. [26] analyzed the Android API for possible sources and sinks of private data and thus identified exemplary privacy policies. All the existing detection methods fundamental cause significant false alarms because

they cannot differentiate legitimate use of sensitive data from authentic privacy leakage. Though effective in terms of privacy protection, these approaches did not attempt to preserve the system usability.

2.4.2 Privacy Leak Mitigation

Based on TaintDroid, Hornyack et al. [19] proposed AppFence to further mitigate privacy leaks. When TaintDroid discovers the data dependency between source and sink, AppFence enforces privacy policies, either at source or sink, to protect sensitive information. At source, it may provide the app with fake information instead of the real one; at sink, it can block sending APIs. To take usability into consideration, the authors proposed multiple access control rules and conducted empirical studies to find the optimal policies in practice.

The major limitation of AppFence is the lack of efficiency. AppFence requires modifications in the Dalvik virtual machine to track information flow and incurs considerable performance overhead (14 % on average according to TaintDroid [12]). Besides, the deployment is also challenging. For one thing, end users have to re-install the operating system on their mobile device to enable AppFence. For another, once the Android OS upgrades to a new version, AppFence needs to be re-engineered to work with the novel mechanisms.

2.4.3 Information Flow Control

Though AppFence is limited by its efficiency and deployment, it demonstrates that it is feasible to leverage Information-Flow Control (IFC) technique to address the privacy leakage problem in Android apps. In fact, IFC has been studied on different contexts. Chandra and Franz [5] implement an information flow framework for Java virtual machine which combines static analysis to capture implicit flows. JFlow [30] extends the Java language and adds statically-checked information flow annotations. Jia et al. [21] proposes a component-level runtime enforcement system for Android apps. duPro [32] is an efficient user-space information flow control framework, which adopts software-based fault isolation to isolate protection domains within the same process. Zeng et al. [49] introduces an IRM-implementation framework at a compiler intermediate-representation (IR) level.

2.5 Text Analytics for Android Security

Recently, efforts have been made to study the security implications of textual descriptions for Android apps. WHYPER [33] used natural language processing technique to identify the descriptive sentences that are associated to permissions

requests. It implemented a semantic engine to connect textual elements to Android permissions. AutoCog [36] further applied machine learning technique to automatically correlate the descriptive scripts to permissions, and therefore was able to assess description-to-permission fidelity of applications. These studies demonstrate the urgent need to bridge the gap between the textual description and security-related program semantics.

2.5.1 Automated Generation of Software Description

There exists a series of studies on software description generation for traditional Java programs. Sridhara et al. [40] automatically summarized method syntax and function logic using natural language. Later, they [41] improved the method summaries by also describing the specific roles of method parameters and integrating parameter descriptions. They presented heuristics to generate comments and describe the specific roles of different method parameters. Further, they [42] automatically identified high-level abstractions of actions in code and described them in natural language and attempted to automatically identify code fragments that implement high level abstractions of actions and express them as a natural language description. In the meantime, Buse [4] leveraged symbolic execution and code summarization technique to document program differences, and thus synthesize succinct human-readable documentation for arbitrary program differences. Moreno et al. [29] proposed a summarization tool which determines class and method stereotypes and uses them, in conjunction with heuristics, to select the information to be included in the class summaries. The goal of these studies is to improve the program comprehension for developers. As a result, they focus on documenting intra-procedural program logic and low-level code structures. On the contrary, they did not aim at depicting high-level program semantics and therefore cannot help end users to understand the risk of Android apps.

References

1. Aafer Y, Du W, Yin H (2013) DroidAPIMiner: mining API-level features for robust malware detection in android. In: Proceedings of the 9th international conference on security and privacy in communication networks (SecureComm)
2. Arp D, Spreitzenbarth M, Hübner M, Gascon H, Rieck K (2014) Drebin: efficient and explainable detection of android malware in your pocket. In: Proceedings of the 21th annual network and distributed system security symposium (NDSS)
3. Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Outeau D, McDaniel P (2014) FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation (PLDI)
4. Buse RP, Weimer WR (2010) Automatically documenting program changes. In: Proceedings of the IEEE/ACM international conference on automated software engineering (ASE)

5. Chandra D, Franz M (2007) Fine-grained information flow analysis and enforcement in a java virtual machine. In: Proceedings of the 23rd annual computer security applications conference (ACSAC)
6. Chen KZ, Johnson N, D'Silva V, Dai S, MacNamara K, Magrino T, Wu EX, Rinard M, Song D (2013) Contextual policy enforcement in android applications with permission event graphs. In: Proceedings of the 20th annual network and distributed system security symposium (NDSS)
7. Cui W, Peinado M, Wang HJ (2007) Shieldgen: automatic data patch generation for unknown vulnerabilities with informed probing. In: Proceedings of 2007 IEEE symposium on security and privacy
8. Davi L, Dmitrienko A, Sadeghi AR, Winandy M (2011) Privilege escalation attacks on android. In: Proceedings of the 13th international conference on Information security. Berlin/Heidelberg
9. Davis B, Sanders B, Khodaverdian A, Chen H (2012) I-ARM-Droid: a rewriting framework for in-app reference monitors for android applications. In: Proceedings of the mobile security technologies workshop
10. Egele M, Kruegel C, Kirda E, Vigna G (2011) PiOS: detecting privacy leaks in iOS applications. In: Proceedings of NDSS
11. Enck W, Ongtang M, McDaniel P (2009) On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on computer and communications security (CCS)
12. Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN (2010) TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX symposium on operating systems design and implementation (OSDI)
13. Enck W, Ocateau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: Proceedings of the 20th USENIX Security Symposium
14. Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E (2011) Permission re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX security symposium
15. Gibler C, Crussell J, Erickson J, Chen H (2012) AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: Proceedings of the 5th international conference on trust and trustworthy computing
16. Grace M, Zhou Y, Wang Z, Jiang X (2012) Systematic detection of capability leaks in stock android smartphones. In: Proceedings of the 19th network and distributed system security symposium
17. Grace M, Zhou Y, Zhang Q, Zou S, Jiang X (2012) RiskRanker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th international conference on mobile systems, applications and services (MobiSys)
18. Hanna S, Huang L, Wu E, Li S, Chen C, Song D (2012) Juxtap: a scalable system for detecting code reuse among android applications. In: Proceedings of the 9th international conference on detection of intrusions and malware, and vulnerability assessment (DIMVA)
19. Hornyack P, Han S, Jung J, Schechter S, Wetherall D (2011) These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of CCS
20. HP Fortify Source Code Analyzer (2016) <http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/>
21. Jia L, Aljuraidan J, Fragkaki E, Bauer L, Stroucken M, Fukushima K, Kiyomoto S, Miyake Y (2013) Run-time enforcement of information-flow properties on android (extended abstract). In: Computer Security—ESORICS 2013: 18th European symposium on research in computer security
22. Kim J, Yoon Y, Yi K, Shin J (2012) Scandal: static analyzer for detecting privacy leaks in android applications. In: Mobile security technologies (MoST)
23. Lin Z, Jiang X, Xu D, Mao B, Xie L (2007) AutoPAG: towards automated software patch generation with source code root cause identification and repair. In: Proceedings of the 2nd ACM symposium on information, computer and communications security

24. Livshits B, Jung J (2013) Automatic mediation of privacy-sensitive resource access in smartphone applications. In: Proceedings of the 22th USENIX security symposium
25. Lu L, Li Z, Wu Z, Lee W, Jiang G (2012) CHEX: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM conference on computer and communications security (CCS)
26. Mann C, Starostin A (2012) A framework for static detection of privacy leaks in android applications. In: Proceedings of the 27th annual ACM symposium on applied computing
27. Martin M, Livshits B, Lam MS (2005) Finding application errors and security flaws using PQL: a program query language. In: Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications
28. McAfee Labs Threats report Fourth Quarter (2013) <http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2013.pdf>
29. Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K (2013) Automatic generation of natural language summaries for java classes. In: Proceedings of the 2013 IEEE 21th international conference on program comprehension (ICPC)
30. Myers AC (1999) JFlow: practical mostly-static information flow control. In: Proceedings of the 26th ACM symposium on principles of programming languages (POPL)
31. Newsome J (2006) Vulnerability-specific execution filtering for exploit prevention on commodity software. In: Proceedings of the 13th symposium on network and distributed system security (NDSS)
32. Niu B, Tan G (2013) Efficient user-space information flow control. In: Proceedings of the 8th ACM symposium on information, computer and communications security
33. Pandita R, Xiao X, Yang W, Enck W, Xie T (2013) WHYPER: towards automating risk assessment of mobile applications. In: Proceedings of the 22nd USENIX conference on security
34. Peng H, Gates C, Sarma B, Li N, Qi Y, Potharaju R, Nita-Rotaru C, Molloy I (2012) Using probabilistic generative models for ranking risks of android apps. In: Proceedings of the 2012 ACM conference on computer and communications security (CCS)
35. Privacy Blocker (2016) <http://privacytools.xeudoxus.com/>
36. Qu Z, Rastogi V, Zhang X, Chen Y, Zhu T, Chen Z (2014) Autocog: measuring the description-to-permission fidelity in android applications. In: Proceedings of the 21st conference on computer and communications security (CCS)
37. Rastogi V, Chen Y, Jiang X (2013) DroidChameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the 8th ACM symposium on information, computer and communications security (ASIACCS)
38. Razmov V, Simon D (2001) Practical automated filter generation to explicitly enforce implicit input assumptions. In: Proceedings of the 17th annual computer security applications conference
39. Sidiroglou S and Keromytis AD (2005) Countering network worms through automatic patch generation. *IEEE Secur Priv* 3:41–49
40. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM international conference on automated software engineering (ASE)
41. Sridhara G, Pollock L, Vijay-Shanker K (2011) Generating parameter comments and integrating with method summaries. In: Proceedings of the 2011 IEEE 19th international conference on program comprehension (ICPC)
42. Sridhara G, Pollock L, Vijay-Shanker K (2011) Automatically detecting and describing high level actions within methods. In: Proceedings of the 33rd international conference on software engineering (ICSE)
43. T.J. Watson Libraries for Analysis (2015) http://wala.sourceforge.net/wiki/index.php/Main_Page
44. Wu C, Zhou Y, Patel K, Liang Z, Jiang X (2014) AirBag: boosting smartphone resistance to malware infection. In: Proceedings of the 21th annual network and distributed system security symposium (NDSS)

45. Xu W, Bhatkar S, Sekar R (2006) Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: Proceedings of the 15th conference on USENIX security symposium
46. Xu R, Sadi H, Anderson R (2012) Aurasium: practical policy enforcement for android applications. In: Proceedings of the 21th USENIX security symposium
47. Yan LK, Yin H (2012) DroidScope: seamlessly reconstructing OS and Dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX security symposium
48. Yang Z, Yang M, Zhang Y, Gu G, Ning P, Wang XS (2013) AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 20th ACM conference on computer and communications security (CCS)
49. Zeng B, Tan G, Erlingsson U (2013) Strato: a retargetable framework for low-level inlined-reference monitors. In: Proceedings of the 22th USENIX security symposium
50. Zhang C, Wang T, Wei T, Chen Y, Zou W (2010) IntPatch: automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Proceedings of the 15th European conference on research in computer security
51. Zhang Y, Yang M, Xu B, Yang Z, Gu G, Ning P, Wang XS, Zang B (2013) Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 20th ACM conference on computer and communications security (CCS)
52. Zhou Y, Jiang X (2012) Dissecting android malware: characterization and evolution. In: Proceedings of the 33rd IEEE symposium on security and privacy. Oakland
53. Zhou Y, Jiang X (2013) Detecting passive content leaks and pollution in android applications. In: Proceedings of the 20th network and distributed system security symposium
54. Zhou Y, Wang Z, Zhou W, Jiang X (2012) Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: Proceedings of 19th annual network and distributed system security symposium (NDSS)

Android Application Security

A Semantics and Context-Aware Approach

Zhang, M.; Yin, H.

2016, XI, 105 p. 37 illus., 29 illus. in color., Softcover

ISBN: 978-3-319-47811-1