2016 IEEE Symposium on Security and Privacy

SoK: Lessons Learned From Android Security Research For Appified Software Platforms

Yasemin Acar*, Michael Backes*†, Sven Bugiel*, Sascha Fahl*, Patrick McDaniel‡, Matthew Smith§
*CISPA, Saarland University, †MPI-SWS, ‡Pennsylvania State University, §University of Bonn

Abstract—**Android security and privacy research has boomed in recent years, far outstripping investigations of other appified platforms. However, despite this attention, research efforts are fragmented and lack any coherent evaluation framework. We present a systematization of Android security and privacy research with a focus on the appification of software systems. To put Android security and privacy research into context, we compare the concept of appification with conventional operating system and software ecosystems. While appification has improved some issues (e.g., market access and usability), it has also introduced a whole range of new problems and aggravated some problems of the old ecosystems (e.g., coarse and unclear policy, poor software development practices). Some of our key findings are that contemporary research frequently stays on the beaten path instead of following unconventional and often promising new routes. Many security and privacy proposals focus entirely on the Android OS and do not take advantage of the unique features and actors of an appified ecosystem, which could be used to roll out new security mechanisms less disruptively. Our work highlights areas that have received the larger shares of attention, which attacker models were addressed, who is the target, and who has the capabilities and incentives to implement the countermeasures. We conclude with lessons learned from comparing the appified with the old world, shedding light on missed opportunities and proposing directions for future research.**

I. Introduction

Over the last couple of years, the *appification* of software has drastically changed the way software is produced and consumed and how users interact with computer devices. With the rise of web and mobile applications, the number of apps with a highly specialized, tiny feature set drastically increased. In appified ecosystems, there is an app for almost everything, and the market entrance barrier is low, attracting many (sometimes unprofessional) developers. Apps are encouraged to share features through inter-component communication, while risks are communicated to users via permission dialogs. Based on the large body of research available for Android as the pioneer of open source appified ecosystems, we center this paper's scope on Android security and privacy research. This choice allows us to focus on the dominant appified ecosystem with a large real-world deployment: Android.

Motivation for a Systematization of Android/Appification Security. The large body of literature uncovered a myriad of appification-specific security and privacy challenges as well as countermeasures to face these new threats. As with all new fields of endeavor, there is no unified approach to research. As a consequence, efforts over the last half decade necessarily pioneered ways to examine and harden these systems. A problem with this approach is that there are lots of fragmented efforts to improve security and privacy in an appified platform, but no unified framework or understanding of the ecosystem as a whole. Therefore, we believe that it is time to systematize the research work on security and privacy in appified platforms, to offer a basis for more systematic future research.

Challenges and Methodology of the Systematization. While the fragmentation of the Android security research is our main motivation, it is at the same time our biggest challenge. Contributions to this research field have been made in many different areas, such as static code analysis, access control frameworks and policies, and usable security for end users as well as app and platform developers. To objectively evaluate and compare the different approaches, our first step will be to create a common understanding of the different security and privacy challenges and a universal attacker model to express these threats. Security solutions are by default designed with a very specific attacker model in mind. We found that in most Android research, this attacker model has been only implicitly expressed. However, to understand the role of a (new) approach within the context of Android's appified ecosystem, it is also important to understand which attacker capabilities it does *not* cover and how different approaches can complement each other. By studying the evaluation details of many representative approaches from the literature, we create a unified understanding of attacker capabilities. This forms the basis for analyzing the security benefits of different solutions and lays the groundwork for comparing approaches with respect to their role in the overall ecosystem.

One insight from our analysis of the challenges in Android's appified ecosystem, is that some security issues are new and unique to Android, as caused by the appification paradigm or the result of design decisions of its architects. Other well-known problems are aggravated by appification, while many security issues are lessened or solved by the appification paradigm. Such understanding is key to transcending Android to develop a broader picture of the future of software systems and the environments they will be placed in.

In particular, the tight integration of many non-traditional actors in the appified ecosystem creates interesting problems as well as opportunities. Platform developers, device vendors, app markets, library providers, app developers, app publishers, toolchain providers and end users all have different capabilities and incentives to contribute (in)securely to the ecosystem. Our systematization makes the important contribution of showing how previous research has interacted with these actors, identifying contributing factors to our research community's work creating a real-world impact.

Based on our systematization of this knowledge, we draw

DOI 10.1109/SP.2016.33

IEEE computer society

lessons learned from our community's security research that provide important insights into the design and implementation of current and future appified software platforms. We also create an overview of which areas have received focused attention and point out areas where research went astray. Finally, we address underrepresented areas that could benefit from or require further analysis and effort.

Please note that we are not discussing plausible problems and benefits of research solutions for adoption by Google or other vendors. Such factors can be manifold, such as technical reasons (e.g., backwards compatibility), business decisions (e.g., interference with advertisement networks), protection of app developers (e.g., intrusion of application sandboxes), or usability aspects. However, without concrete first-hand knowledge, any such discussion would merely result in speculation, which we do not consider a tangible contribution of a systematization of knowledge.

Systematization Methodology

There is a huge body of research work on Android security with (conservatively) over 100 papers published. Since we aim to systematize this research as opposed to offering a complete survey [1], we extracted key aspects and key papers to create a foundation for our systematization. The focus of our systematization is on security issues and challenges in the context of appification and the app market ecosystem. We include both offensive works (i. e. papers that uncovered new security issues or classes of attacks) as well as defensive ones (i. e. papers that focus on countermeasures or new security frameworks). However, we do not focus on malware on appified platforms, as this has been dealt with in prior work [2]. We also exclude hardware-specific or other low level problems on mobile platforms, such as CPU side-channels, differential power analysis, or base-band attacks, which are independent from appification.
We selected the research based on the following criteria:

- *Unique/Pioneering*—Security issues which are unique to the Android ecosystem, i.e. never been seen before.
- *Aggravated*—Security issues which have greater impact on an appified ecosystem than on traditional computing.
- *Attention*—Research on aspects that received more attention (i. e. many papers dealt with this specific aspect or the papers received high citation counts).
- *Impact*—Security research that affected a large number of users (or devices).
- *Scope*—Security issues which involve a large fraction of the appified world's actors. We include these issues since they are particularly hard to fix.
- *Open Challenges*—Research worked on issues or countermeasures that remain "unfinished" and highlight interesting and important areas of future work.

In the following, we systematize the research using the above rubric, extract a unified attacker model and evaluate the work both in terms of content and also on its placement within the Android ecosystem. We identify actors that are responsible for the problems, would benefit from solutions, and/or have the capability to implement and deploy them.

II. Problem and Research Areas

To identify important problem and research areas, we compare aspects of traditional software ecosystems with appified platforms, mainly focusing on Android.

A. Conventional Software Ecosystem vs. Appified Platform (Android)

We start our systematization by categorizing and summarizing key security challenges and issues that have been identified in the literature in both conventional software ecosystems and the appified world. Our intention for systematizing the key security challenges is to provide a systematic approach to help security researchers understand the (old and new) challenges that have been identified and to lay the foundation for a discourse on addressing these challenges.

1) Defining the Access to Resources: Controlling access to resources on a computer system requires 1) accurate definition of the security principals and protected resources in the system; 2) a non-bypassable and tamper-proof validation mechanism for any access (*reference monitor*); and 3) a sound security policy that governs, for all requested accesses in any system state, whether access is allowed or should be denied. Android deviates from conventional OSes in all three aspects:

a) System Security Principals: Conventional systems are primarily designed as multi-user systems with human users that have processes executing on their behalf. A small number of dedicated user IDs is assigned to system daemons and services that do not execute on behalf of a human user.

Appified security models build on the classic multi-user system: not only is the human user of the system considered a principal, but in fact all app developers that have their app(s) installed on the system are considered as security principals. Developers are represented by their app, which receives a distinct user ID (UID), exactly like the pre-installed system apps receive a UID. In recent Android versions with multi-(human)-user support, the traditional UID scheme is further extended: the UID is now a two-dimensional matrix that identifies the combination of the app UID (i.e., developer) and human user ID under which the app is currently running.

b) Implementation of the Reference Monitor: Conventionally, reference monitoring is typically managed by the OS, e.g., the file system and network stack, so that user processes can build their access control on top.

Appified ecosystems also use the OS for low-level access control. However, the extensive application frameworks on top of which apps are deployed provide a different interface: following the paradigm of IPC-based privilege separation and compartmentalization in classical high assurance systems, security- and privacy-critical functionality is consolidated into dedicated user-space processes. Exposed IPC interfaces enforce access control on calling processes.

c) Security Policy: In conventional software systems, multiple privilege level(s) for a process are defined: Processes can run as superuser (root), system services, with normal user privileges, guest privileges, and so on. All processes running under a certain privilege level share the same set of permissions and may access the same set of resources.

Modern appified ecosystems make a clearer distinction between system and third-party apps: Direct access to security- and privacy-sensitive resources (e.g., driver interfaces or databases) is only permitted to selected applications and daemons of the application framework. This policy is implemented, as in the conventional platforms, in the OS access control policies (i.e., discretionary and mandatory access control). However, system apps may request access to permissions that are not available to third-party apps. Third-party apps have by default no permissions set, but may request their permissions from a set commonly available to all third-party apps.

2) Sharing Functionalities: In conventional operating systems, third-party apps are usually self-contained and heavily used to incorporate external functionalities as libraries (e.g. the OpenSSL library to make TLS available in a program).

In addition to third-party libraries, in the appified world, apps also share functionality through inter-component communication (ICC), i.e., by providing a Service that can be accessed through Intents or persistent IPC connections. ICC is heavily used to access system apps such as the map, phone, or Play app, but also popular third-party services, e.g., as offered by the Facebook and Twitter apps.

3) Software Distribution: Conventionally, software is distributed in a decentralized way: It can be downloaded from websites, purchased in online stores or shipped on physical media such as USB sticks or CDs. Software comes either in compiled binaries or, in case of open source software, as source code that needs to be compiled before installation.

Appified ecosystems often make use of centralized stores that distribute software/apps. These app stores allow developers to upload and distribute their software in a highly organized way. The app markets provide search, feedback and review interfaces for users and allow for centralized security mechanisms that can be enforced by the markets directly. We distinguish between commercial app markets such as Google Play and central software repositories that are widely used in different Linux distributions. In addition to simply distributing software by streamlining the process of searching and installing apps, commercial app markets have additional responsibilities such as billing, DRM (e.g., forward locking on Android) and in-app purchasing.

4) Software Engineering:

a) Development Process: Previously, single developers/-companies developed software and in many cases distributed it themselves. They followed agreed-on rules (e.g., IDE, libraries, or frameworks to use) and could outsource in a regulated way to contracted (sub-)companies. In appified ecosystems, a chain of actors is responsible for the distribution of software, which is much more loosely coupled than the more stringent traditional development chains: The original developer, (often) a publisher, and increasingly development frameworks are involved.

b) Programming Environment: In conventional operating systems, developers can choose what programming language they want to use (within the design space that the project leaves them), and a wide range of programming languages and frameworks are usually available to implement software. Appified ecosystems dictate programming languages and frameworks to enforce compatibility with their application framework and hence robustness of the deployed applications. Android developers, for instance, are required to use Java and the Android SDK/NDK. App creators play a crucial role in modern appified ecosystems: They provide easy-to-use clickable interfaces to produce software that can be run on multiple platforms.

5) Present Classes of Programming Errors: Programming errors, such as logic errors and run-time errors, are the dominant sources of software vulnerabilities in conventional software ecosystems. While recent years have demonstrated that they are also present in mobile platforms with the same devastating effects, the API-dependent design of apps has introduced a new range of problems into the appified world as a direct consequence of misuse of programming APIs of the surrounding application framework. This differs from the traditional ecosystem, where this class of errors is limited mostly to library APIs, since the application framework API is a necessity to make apps operational.

6) Webification: In conventional software ecosystems, software is mainly self-contained and its primary functionality does not depend on the availability of remote resources such as web services. The appification paradigm has seen a shift towards increasingly web service-oriented architectures that depend on server backends to provide their promised functionality. At the end of the spectrum are apps that consist merely of a webview component that appears to be local app logic, but in fact is not much more than a restricted web browser for the service's backend web servers.

7) Software Update Mechanisms: Conventional OS updates are centrally organized, while the updating process for third-party software takes, in contrast, a greater effort: Every program needs to be updated (and hence, often started and restarted) separately. Only systems with a central software distribution channel improve on this situation (e.g., Linux distributions). The situation for updates in appified ecosystems is currently the exact opposite. Fragmentation is a huge issue in appified ecosystems, such as Android, and impedes the OS update process. As many different network providers and device vendors customize parts of the operating system, they need to manage OS updates on their own, resulting in lengthy and complicated update procedures. As a result, many Android devices do not receive OS updates at all. In contrast, app updates are straightforward and fast, as centralized app stores push updates immediately to their users.

III. Android/Appified Ecosystem

As an example for appification, we provide an overview of the Android ecosystem, the actors involved and their impact on the ecosystem's overall security. We use Figure 1 as our reference to introduce the actors and their interaction patterns.

A. Ecosystem Overview

At the core of appification ecosystems are the *app developers*, producing the millions of apps available for the *end users*. The number of Android app developers is vastly larger than for the traditional desktop software ecosystem. For instance, in the current Play market[1] roughly 460,000 distinct developer accounts have published applications, where an account can also belong to an entire company or team of developers. These app developers rely on the rich APIs of the platform SDK, which is provided by the *platform developers*. These APIs provide access to core functionalities (e.g., telephony, connectivity and sensors like accelerometers) as well as to user data (e.g., contact management, messaging, picture gallery).

Developers can request access to those functionalities by requesting permissions in their app's manifest file (e.g. the `CONTACTS` permissions grants access to the user's address book). End users are presented permission dialogs at install time. Those dialogs present all the permissions previously requested by a developer and inform users about an app's resource access. Since version 6 (*Marshmallow*), Android also introduced, like iOS has done several iterations before, the concept of dynamic permissions: a small subset of all permissions are granted by the user at runtime when an app requests access to protected interfaces instead of statically at install time, and those selected permissions can also be revoked again by the user. It is also possible for developers to define custom permissions that can grant access to their app's functionality to other apps written by the same developer, system apps, or all apps installed on the device.

Android apps are composed of Java code (compiled to bytecode format for the CPUs of mobile platforms) and of native code in the form of C/C++ shared libraries. *Library providers* such as *advertisement networks* support developers in creating ad-supported apps by offering dedicated *ad libraries* that apps can rely on, thus firmly integrating the ad library in the final application package. Many apps connect to *web-services* (e.g., cloud-based services or other backends) and use web-technologies such as HTML, CSS and Javascript. This move to web apps is typical for the appification paradigm.

Typically for the shift to appification is the way monetization works: App developers can sell their apps to end users for fixed one-time prices (using central app stores such as Google Play), they can collaborate with *advertisement networks* by displaying advertisements in their apps and receiving shares of the advertisement revenues, or they can offer in-app purchases, e.g., users can buy additional features of the app. Those options are not mutually exclusive, but conventionally paid apps refrain from displaying ads. Together they lower the economic burden on developers and streamline the process of purchasing and installing apps for end users [3].

TABLE I
ALL ACTORS IN THE ECOSYSTEM AND THE IMPACT OF THEIR SECURITY DECISIONS ON THE REMAINING ACTORS.

Actor	OS Developer	Hardware Vendor	Library Provider	Software Developer	Toolchain Provider	Software Publisher	Software Market	End User
OS Developer	■	◐	●	●	◐	●	●	●
Hardware Vendor	○	■	●	●	○	○	○	●
Library Provider	○	○	■	●	○	○	○	●
Software Developer	○	○	◐	■	○	○	○	●
Toolchain Provider	○	○	○	●	■	○	○	◐
Software Publisher	○	○	◐	◐	○	■	◐	●
Software Market	○	○	◐	◐	○	○	■	●
End User	○	○	○	○	○	○	○	■

● = fully applies; ◐ = partly applies, ○ = does not apply at all.

Unlike other current appified ecosystems, Android allows (and actually encourages) inter-component communication (ICC), which prompts developers to divide their apps into smaller parts (e.g., plugins) and allows them to act as service providers (e.g., Facebook app, Play app, etc.). Technically, ICC is based on the Linux kernel's inter-process communication—primarily via a new IPC mechanism called *Binder*. However, since logical communication occurs between application components such as databases, user interfaces, and services, this Android-specific IPC has been coined as *Inter-Component Communication* in the literature [4].

B. Involved Actors

Software ecosystems involve a number of actors that each have their own rights and duties, which differ between appified and conventional ecosystems in some aspects. We differentiate these actors as groups of ecosystem participants, describe their primary task(s), their power to influence the security and privacy of the ecosystem with their decisions, and then give concrete examples of each class of actors. Table I illustrates the different actors, their influence on the ecosystem's security and privacy, and their interaction with each other.

Although feedback loops can be established between any number of actors, in the following discussion we focus on the potential *direct* impact of a security decision made by one user on all other actors. We do not consider *indirect* impact, e.g., when users protest against or boycott certain apps and thus force app or platform developers to react.

1) Platform Developers: Platform developers are responsible for providing the Android Open Source Platform (AOSP). They make basic system and security decisions and *all other actors* build on their secure paradigms. Library providers and app developers are bound to the provided SDK, and app markets have to rely on Android's open approach (instead of, for

[1] Approximately 1.5 million free apps crawled in February 2016.

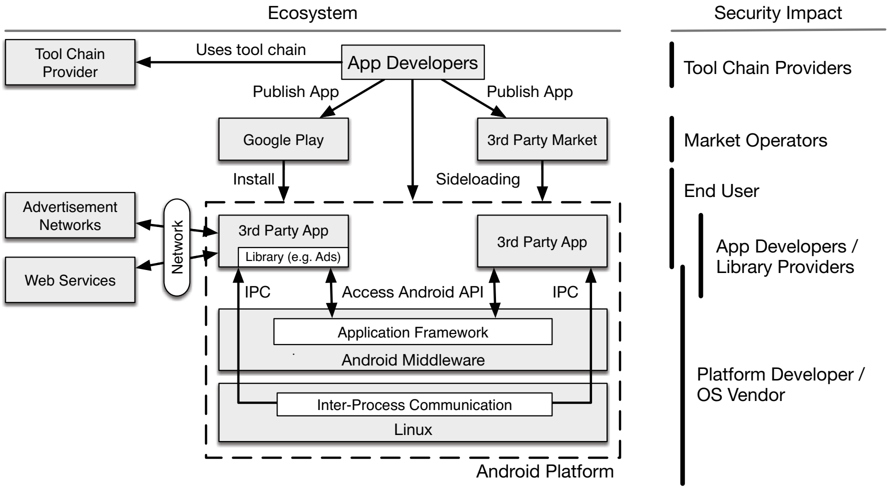

Fig. 1. The Android ecosystem: Actors and their impact on the ecosystem's security.

example, Apple's walled-garden ecosystem). An exception is that device vendors can implement their own security decisions and need not adhere to Android's paradigms. In reality, though, they mostly build upon the provided foundations.

2) Device Vendors: Device vendors adopt the AOSP and customize it for their different needs. A variety of device vendors currently share the market for mobile devices using Android [5]. Besides adaptation of the basic Android software stack to the vendor-specific hardware platforms, vendors customize in order to distinguish their Android device from their competitors'. Thus, many versions of vendor-specific apps and modified versions of Android's original user interface are being distributed with Android-based platforms. The impact of device vendors on the ecosystem's security is significant: Although, naturally, their customizations only affect their customers, this user-base can be large in case of big vendors such as Samsung or HTC. Device vendors can adopt security decisions from the platform developers or add their own solutions (cf. Samsung KNOX [6]) on which library and app developers can build. However, device vendors cannot change the way apps are published in markets, which is why their impact on publishers and markets is very limited—e.g. they could not enforce CA-signed instead of self-signed certificates for app signing practices without breaking Android's guidelines.

3) Library Providers: Based on the platform's API, library providers build their own APIs to offer new features such as *ad services* or to make the use of (possibly unnecessarily) complicated platform APIs easier for app developers. Libraries exist for UI components (they can but need not be attached to network tasks) as well as for ads or crash reports. Library developers have the power to make all apps that include them either more or less secure. Library developers suffer or benefit from security decisions made by platform developers and device vendors. However, their decisions do not affect the platform security in general. Their positive/negative security decisions propagate to app developers who choose to use their libraries—they can, for example, wrap badly designed programming interfaces from platform developers. Their decisions affect neither app publishers nor markets directly. Typically, library providers offer *ad services*, *networking features* or *app usage evaluation features*.

4) App Developers: App developers write apps using the APIs defined by platform developers and of those libraries they choose to include. They can opt to write code themselves or use existing third-party code. In theory, they can make essential contributions to security. In practice, they make unsafe choices and implement features in the least laborious way, which is frequently not the most secure choice.

While app developers can break secure default interfaces provided by platform developers/device vendors (e.g. crypto primitive API misuse), this has no effect on the platform security in general. Their decisions neither affect app publishers nor markets directly. Still, app developers may impact libraries' security (e.g. as fraud is a frequently evaluated issue).

5) Toolchain Providers: Toolchain providers offer helpful tools for app developers (e.g. the Eclipse ADT for Android app development). They can implement many analysis tools that help discover API misuse. Toolchain providers can fix some weaknesses introduced by platform providers and device vendors (e.g. confusing permission descriptions, or hard-to-use APIs). All app developers and their users benefit/suffer from good/bad toolchain provider support.

6) App Publishers: App publishers are professional service providers that help developers publish their apps to certain markets. They receive either binary or source code, add certain properties like ads, and distribute the app to one or more app markets. In theory, they can run preliminary analyses on the code and report or fix bugs, as well as filter malware. If app signing is delegated to the app publishers, they could also surreptitiously insert malicious code. Several app publishers maintain substantial numbers of apps [7] and thereby may substantially impact markets' security. Hence, a single publisher's impact on the ecosystem's security is rather impressive.

7) App Markets: App markets—Google Play is the most popular one—distribute apps from developers to the end users. Users as well as app developers rely on them to make sure that the apps are distributed in a consistent, unchanged, reliable, and benign way. In theory, app markets have the potential to find not only malware, but also buggy and unsafe code. To do this, they can apply various kinds of security analyses techniques—such as static or dynamic code analysis—on all apps they distribute. For example, Google Play runs supposedly multiple tests on apps prior to distribution, including static/dynamic analysis and machine learning [8]. However, they do not run deeper checks to detect dangerous misuse of the Android API. No app market runs (theoretically possible) runtime tests, nor do they exclude apps signed with the same key corresponding to different developers.

8) Users: Users are app consumers in the ecosystem. They can make the decision to install (non-pre-installed) apps, and have to confirm the permissions that apps request. They are the most likely target of attacks. In theory, they can make safe choices, as well as choose not to use important credentials. However, a single user's impact on the ecosystem's security is negligible. Users as a group have to rely on security decisions made by all other actors in the ecosystem.

C. Global Attacker Model

We provide a taxonomy for *attacker capabilities* on Android. This taxonomy reflects the *threat models* we extracted during our systematization in Section IV and helps to later on compare proposed countermeasures.

When considering the attacker capabilities, we had the options to order them *across capability categories* or *within categories*. We decided to order them *within categories*, since our categories depend on too many distinct factors to be comparable and since we base our systematization on those categories. For instance, a user connecting frequently to public Wi-Fi access points is susceptible to network attacks, but this behavior does not influence other capability categories like, e.g., piggybacking apps. We order the attacker capabilities vertically, i.e., we rate the power of attackers in specific capability categories. We use the following semantics to note attacker capabilities in each category: Solid circles (●) denote strong capabilities corresponding to a weak attacker model. Half-filled circles (◐) denote common attacker capabilities, while hollow circles (○) describe the absence of any capability in the category, strengthening the attacker model.

Next, we introduce our categories for attacker capabilities, informally define the exact capabilities attackers may have in each category, and explain our ordering of those capabilities.

C1—*Dangerous permissions:* The attacker has code running on the victim device, which has been granted *dangerous* permissions (●) that give access to privacy sensitive user data or control over the device that can negatively impact the user. Dangerous permissions must be explicitly granted by the user during app installation. We assume *normal* permissions (◐) when the attacker has been granted only permissions that are of lower risk and automatically granted by the system.

C2—*Multiple apps:* Attacker-controlled apps are running on the user device. Full capability indicates that the attacker has two or more apps running on the victim device (●). This would enable collusion attacks via overt and covert channels. Half-capability (◐) means that only one attacker-controlled app is running on the device. In general, the capability of having at least one app on the user device enables the attacker to engage in ICC with other apps on the device or to scan the local file system to the extent the attacker-controlled apps' permissions allow this.

C3—*Piggybacking apps:* The attacker re-packages other apps and is able to modify the existing code or include new code (●). A limited piggybacking capability (◐) is assumed if the attacker provides code that is intentionally loaded by app developers into their apps (e.g., libraries). Limited piggybacking is assumed to be the weaker capability, because libraries used by developers are hosted by the app (i.e., share the host sandbox) limiting the attacker to the host app's permissions. In contrast, re-packaging apps allows the attacker to request more permissions for the repackaged app.

C4—*Native code:* The attacker has an app containing native code, i.e., shared libraries. This requires having at least one app on the device under control (**C2.◐**). Native code that implements exploit payload, native programs, or zipper/crypto routines for obfuscation are considered as full capability (●). Non-exploit code that still provides the means to modify the app's memory space is assumed as half-capability (◐). Although Android's design permits all apps to contain native code, there are apps that contain none (○).

C5—*Dynamic code loading:* The attacker is able to dynamically load code at run-time (●) into an app (e.g., using the Java reflection API). This requires having at least one app on the device under control (**C2.◐**). Half-capability (◐) is assumed if the attacker can inject code into another, benign but insecure app. Dynamic code loading is assumed to be a stronger capability than code injection, since dynamic loading allows the attacker to use obfuscation techniques to execute the attack surreptitiously.

C6—*Network attacks:* The attacker is capable of modifying/interrupting/forging the Wi-Fi and cellular network communication of the end user device (●). We assume a passive attacker (◐) if the attacker is only able to eavesdrop on the communication. Technically, a network attack can be accomplished as in traditional attacker models by, e.g., setting up a rogue access point or base station. On Android, an attacker can gain the same capability through a malicious VPN app, through which all network traffic of all processes is routed when it is activated by the user. This requires at least **C2.◐**.

TABLE II
SECURITY CHALLENGES IN THE APPIFIED ECOSYSTEM, ACTORS CAPABLE OF REDRESSING THE PROBLEMS, AND ATTACKER CAPABILITIES CONSIDERED IN THREAT MODELS OF DIFFERENT PROBLEM AREAS.

				Causative Actors								Fixable by								Attacker capabilities					
Problem area	**Focus**	**References**	Discussed in Section	A1. Platform Developers	A2. Device Vendors	A3. App Markets	A4. Library Providers	A5. App Developers	A6. App Publishers	A7. Toolchain Providers	A8. End Users	R1. Platform Developers	R2. Device Vendors	R3. App Markets	R4. Library Providers	R5. App Developers	R6. App Publishers	R7. Toolchain Providers	R8. End Users	C1. Sensitive permissions	C2. Multiple apps	C3. Piggybacking apps	C4. Native code	C5. Dynamic code loading	C6. Network attacks
Permission-based Access Control and Least Privilege	Permission Attention and Comprehension by End Users	[9], [10], [11]	IV-A1a	●	○	●	◐	◐	○	○	◐	●	○	●	◐	◐	○	◐	○	●	◐	○	○	○	○
	Permission Comprehension by App Developers	[12], [13], [14], [15], [16], [17]	IV-A1b	●	○	○	◐	◐	○	○	○	●	○	○	○	○	○	◐	○	○	◐	○	○	○	●
	Permission Attention by App Developers	[17], [18], [19], [20], [21], [22], [23], [24], [15]	IV-A1b	◐	○	○	○	●	○	○	○	◐	◐	○	○	●	○	○	○	○	◐	○	○	○	○
	Missing Privilege Separation	[25], [26], [27], [28], [29], [30], [31]	IV-B1a	●	○	○	●	◐	○	○	○	●	○	○	◐	○	○	○	○	●	◐	◐	○	●	○
	Missing Efficacy of Security Apps	[2], [32], [33]	IV-B1b	●	○	○	○	○	○	○	○	●	○	○	○	○	○	○	◐	●	◐	○	●	◐	○
Webification Issues	—	[34], [35], [36], [37], [38], [39]	IV-C1	●	○	○	●	○	○	●	○	●	○	○	●	○	○	●	○	○	◐	○	○	○	●
API Misuse of App Development Frameworks	—	[40], [41], [42], [43], [44], [30], [45]	IV-D1	◐	○	○	●	●	○	○	○	◐	○	◐	●	●	○	◐	○	○	●	○	○	●	●
Software Distribution Channels	App Piracy and Malware Incentives	[46], [47], [32], [48], [49], [50]	IV-E1a	○	○	◐	○	●	○	○	○	○	○	◐	○	○	○	○	◐	○	○	●	○	○	○
	Application Signing Issues	[11], [7]	IV-E1b	●	○	◐	○	●	●	○	○	●	○	◐	○	●	●	○	○	○	○	●	○	○	○
Vendor Customizations and Fragmentation of the Ecosystem	—	[24], [15], [51], [52], [53]	IV-F1	●	●	○	○	○	○	○	○	●	●	○	○	○	○	○	○	●	◐	○	○	○	○

● = fully applies; ◐ = partly applies, ○ = does not apply at all.

TABLE III

Categorization of proposed Android security countermeasures, their potential implementers, and their addressed attacker model.

				Possible implementers								Considered attacker model					
Problem area	**Focus**	**Solution**	**Reference**	R1. Platform Developer	R2. Device Vendor	R3. App Market	R4. Library Provider	R5. App Developer	R6. App Publisher	R7. Toolchain Provider	R8. User	C1. Sensitive permissions	C2. Multiple apps	C3. Piggybacking apps	C4. Native code	C5. Dynamic code loading	C6. Network attacks
Permission evolution (Section IV-A)	System Security Extension	Kirin	[17]	●	●	○	○	○	○	○	◐	●	○	○	○	○	○
		TaintDroid	[54]	●	●	○	○	○	○	○	○	●	◐	○	○	○	○
		Apex	[55]	●	●	○	○	○	○	○	◐	●	◐	○	○	○	○
		Sorbet	[21]	●	●	○	○	◐	○	◐	○	●	◐	○	○	○	○
		QUIRE	[56]	●	●	○	○	◐	○	○	○	○	◐	○	◐	○	○
		IPC Inspection	[20]	●	●	○	○	○	○	○	○	○	◐	○	◐	○	○
		XManDroid	[57]	●	●	○	○	○	○	○	○	●	●	○	◐	○	○
	SDK / Tool-chain Extension	Stowaway	[13]	○	○	○	○	◐	●	●	○	●	○	○	○	○	○
		PScout	[14]	○	○	○	○	◐	●	●	○	●	○	○	○	○	○
		Curbing Permissions	[58]	○	○	○	○	◐	●	●	○	●	○	○	○	○	○
	HCI Modifications	Decision making process	[59]	○	○	●	○	○	○	○	●	●	○	○	○	○	○
		Using personal information	[60]	○	○	●	○	○	○	○	●	●	○	○	○	○	○
	(Meta) Data Analysis	WHYPER	[61]	○	○	●	○	◐	◐	○	◐	●	○	○	○	○	○
		AutoCog	[62]	○	○	●	○	◐	◐	○	◐	●	○	○	○	○	○
		DescribeMe	[63]	○	○	◐	◐	●	●	◐	○	●	○	◐	○	○	○
	User study	Permissions remystified	[64]	●	●	○	○	●	○	○	●	●	◐	○	○	○	○
Permission revolution (Section IV-B)	System Security Extension	Saint	[65]	●	●	○	○	◐	○	○	○	●	◐	○	◐	○	○
		CRePE	[66]	●	●	○	○	○	○	○	◐	●	◐	○	◐	○	○
		TISSA	[67]	●	●	○	○	○	○	○	○	●	◐	○	◐	○	○
		SE Android	[68]	●	●	○	○	○	○	○	○	●	◐	○	●	○	○
		TrustDroid	[69]	●	●	○	○	◐	○	○	○	●	◐	○	●	○	○
		FlaskDroid	[70]	●	●	○	○	○	○	○	○	●	◐	○	●	○	○
		ASM	[71]	●	●	○	○	(◐)‡	○	○	(◐)‡	(●)‡	(◐)‡	○	(●)‡	○	○
		Compac	[72]	●	●	○	○	○	○	○	○	●	◐	●	○	○	○
		AdDroid	[27]	●	●	○	○	◐	◐	○	○	●	◐	◐	○	○	○
		AdSplit	[73]	●	●	○	○	◐	○	○	○	●	◐	◐	○	○	○
		LayerCake	[74]	●	●	○	○	○	○	○	○	○	◐	◐	○	○	○
	Binary modifications	Aurasium	[75]	○	○	◐	○	○	○	◐	●	●	◐	○	◐	●	○
		Dr. Android,Mr. Hide	[76]	○	○	●	○	●	◐	◐	●	●	◐	○	◐	●	○
		I-ARM Droid	[77]	○	○	◐	○	○	◐	◐	●	●	○	○	○	●	○
		AppGuard	[78]	○	○	◐	○	○	◐	◐	●	●	○	○	○	●	○
		Boxify	[79]	○	○	○	○	○	○	○	●	●	◐	○	◐	●	○
Webification (Section IV-C)	System Security Extensions	Morbs	[35]	●	●	○	○	◐	◐	◐	○	○	◐	○	○	○	●
	SDK / Tool-chain Modification	NoFrak	[34]	○	○	○	●	◐	●	○	○	○	○	○	○	○	●
		NoInjection	[38]	○	○	○	●	◐	●	○	○	○	○	○	○	○	●
Programming-induced leakage (Section IV-D)	SDK / Tool-chain Extension	MalloDroid	[40]	○	○	●	○	◐	●	●	○	○	○	○	○	○	●
		CryptoLint	[43]	○	○	●	○	◐	●	●	○	○	◐	○	○	○	◐
		SSL API Redesign	[42]	●	●	○	○	○	○	◐	○	○	○	○	○	○	●
	App Analysis	SMV-Hunter	[41]	○	○	●	○	◐	●	●	○	○	○	○	○	○	●
		CHEX	[22]	○	○	◐	○	●	●	○	○	●	○	○	○	○	○
		SCanDroid	[80]	○	○	●	○	○	◐	○	◐	●	○	◐	○	○	○
		AndroidLeaks	[81]	○	○	●	○	○	◐	○	○	●	○	◐	○	○	○
		FlowDroid	[82]	○	○	●	○	○	◐	○	○	●	○	◐	○	○	○
Software Distribution (Section IV-E)	Market solution	Meteor	[83]	◐	◐	○	●	○	◐	○	◐	●	○	●	○	○	○
		MAST	[84]	○	○	●	○	○	○	○	○	●	○	●	●	●	○
		Application Transparency	[7]	○	○	●	●	○	○	○	○	○	○	●	○	○	○
	(Meta) Data Analysis	DroidRanger	[32]	○	○	●	○	○	●	○	○	○	○	●	●	●	○
		DNADroid	[49]	○	○	●	○	○	●	○	○	○	○	●	○	○	○
		RiskRanker	[85]	○	○	●	○	○	●	○	○	●	○	●	●	●	○
		CHABADA	[86]	○	○	●	○	○	○	○	○	●	○	●	○	○	○
		Collaborative Verification	[87]	○	○	●	○	●	○	○	○	●	○	○	○	○	○
		MassVet	[88]	○	○	●	○	○	○	○	○	●	○	●	○	○	○
	SDK / Tool-chain Extension	AppInk	[89]	○	○	○	○	◐	●	●	○	○	○	●	○	○	○
Software Update Mechanism (Section IV-G)	(Meta) Data Analysis	SecUp	[51]	●	●	○	●	○	○	○	●	(●)†	◐	○	○	○	○

● = actor must implement solution/attacker capability fully addressed; ◐ = actor should/can participate in solution/attacker capability partially addressed
○ = actor not involved/attacker capability not addressed.
† Requests sensitive permissions and attributes defined by a future Android OS version; ‡ Depends on loaded security module

IV. Systematization of Research Areas in Appified Ecosystems

Building on the differences between conventional and appified ecosystems as well as the actor and global threat model of the Android ecosystem, we now identify fields of research that we think need to be systematized, considering a number of representative research papers for each field. We discuss challenges in the respective fields, regarding the global actor model, identifying the involved actors, their respective roles in causing a specific problem and their potential in resolving it. Referring to the global threat model, we summarize the attacker capabilities assumed in the threat models required to exploit the problem areas. Moreover, we present selected, representative Android security countermeasures if available. We do not claim that our systematization is all-encompassing, nor that it includes all problem fields ever identified for Android nor all countermeasures to known problems; however, we took great care to choose a representative selection (see Section I).

A. Permission Evolution

The concept of permission-based access control for privileged resources is one of the cornerstones of Android's security design and has received a lot of attention by the security research community.

1) Challenges: We sub-categorize the identified problems and challenges according to the most affected actors in the ecosystem: the end users and app developers.

a) Permission Comprehension and Attention by End Users: To effectively inform end users about the privacy risks that an app imposes, it is imperative that end users are capable of correctly perceiving the risk of granting the access rights requested by apps. Pioneering work showed that only a very small fraction of users could correctly associate privacy risks with the respective permissions [9]. One potential root cause for this lack of understanding seems to be that permissions communicate resource access, but do not explain how accessed data is processed and distributed [10]. Hence, users tend to underestimate the risks ("the app will not misuse its permissions") or overestimate the risks ("the app will steal all my private information") [9]. A lack of user comprehension of permissions allows attackers to create malicious apps that request all necessary sensitive permissions for their operations (as demonstrated, e.g., by the Geinimi Trojan [90]).

Apps published after Android v6.0 may request a small subset of privacy-related permissions during runtime instead of at installation. Requesting permissions dynamically when they are required by the app should provide users with more contextual information and help them in their decision making process. However, Wijesekera et al. [64] have shown that this desired contextual integrity—i.e., personal information is only used in ways determined appropriate by the users—is not necessarily provided by dynamic permissions and runtime consent dialogs: A majority of privacy-related permission requests occur when the user is not interacting with the requesting application or even with the phone, and, moreover, requests occur at a frequency that prohibits involving the user in every decision making process. As a consequence, users failed to establish the connection between the permission request and the apps' functionality and consent dialogs are only shown during first request to grant access until manually revoked by the users although subsequent permission checks might occur in a different privacy-context than the initial request.

b) Permission Comprehension and Attention by App Developers: Android's security design requires app developers to contribute to platform security by requesting, defining, and properly enforcing permissions in order to retrieve and protect sensitive user data. Thus, even more than for end users, it is imperative that app developers understand permissions and the security tools at their disposal.

Permission Comprehension by App Developers. A number of studies [12], [13], [14], [16], [17] give insight into how app developers comprehend permissions and, in particular, how the SDK supports them in their task to realize least-privileged apps (e.g., considering the stability of the permission set or the extent to which permission-protected APIs are well-documented). Between 30% [13] and 44.8% [12] of the studied apps requested unnecessary permissions, i.e., were over-privileged and in clear violation of the least-privilege principle. Moreover, several apps have been found that request non-existent or even wrong permissions. Even developers of system apps, who have access to the highest privileged and highly dangerous API functions, did not exhibit a significantly better understanding of permissions [15].

To understand the root causes behind the developers' incomprehension of permissions, the studies analyzed the Android API documentation, finding that the API is insufficiently documented and does not identify all permission-protected APIs. Even worse, the documentation also contained errors, e.g., describing the wrong permission required for an API function. Confusing permission names also contribute to these misconceptions. These inconsistencies and the instability of the API impede a clear and well-developed documentation and thereby contribute to the developers' incomprehension of permissions and to confusion about permission usage.

Permission Attention by App Developers. Besides developers' (lack of) comprehension of permissions, the thoughtfulness of developers when enforcing permissions was studied, as well as their level of comprehension of the mechanisms at their disposal to accomplish this task. Although Android's security design incorporates important lessons learned from prior operating system security research [91], the fact that it allows and even encourages differently privileged apps to communicate with each other has piqued the security research community's interest in how this can be exploited by unprivileged apps to escalate their privileges [17], [18], [19], [20], [21], [22], [23], [24], [15]. In particular, various works have identified an increase in failure of app developers to properly protect their app's IPC-exposed (or `exported`) interfaces and to transitively enforce permissions [20]. This opens the attack surface for confused deputy attacks[2] to, e.g., initiate phone calls [17], hijack ICC [19], or exfiltrate sensitive user data [23], [22]. The root cause of many of those identified vulnerabilities is that application components were by default exported to be IPC-callable and thus require that the developers either explicitly protect them with permissions or hide the components. As indicated by the uncovered vulnerabilities, most developers are unaware of these conditions. To phrase this in the terms of Saltzer's and Schroeder's secure design principles [91]: Android failed to implement fail-safe defaults.

2) Countermeasures: Recent changes [94] in the default installer app for Google Play aim to improve permission perception for users. Installers present permissions with low granularity in groups, while some commonly requested permissions, like `INTERNET`, are not presented at all anymore. This shift in permission presentation can be viewed as a pure user experience decision, not as an enabler of user comprehension.

Research has made several suggestions to enhance the usability of permissions for both end users and developers: Kelley et al. [59] propose to enrich permission dialogs with more detailed privacy-related information to help users make a more informed decision. Porter Felt et al. [95] propose making the permission-granting mechanism dependent on the kind of permission that is requested, e.g., auto-granting non-severe permissions with reversible side-effects, trusted UI for user-initiated or alterable requests, or confirmation dialogs for non-alterable, app-initiated requests that need immediate approval. A concrete realization of trusted UI are access control gadgets by Roesner et al. [96] that allow a user-driven delegation of permissions to apps whenever such widgets can be effectively integrated into the apps' workflows. Wijesekera et al. [64] suggest more intelligent systems that learn about their users' privacy preferences and only confront users with consent dialogs when a permission request is unexpected for the user. This consent dialog should provide sufficient contextual cues for users, e.g., clearly indicating the app requesting the access to protected resources as well as clearly communicating why the resource is accessed. Liu et al. [97] propose eliminating the burden of understanding the enormous list of permissions by using a limited set of privacy profiles including certain permissions instead; and Felt et al. [13] propose to improve API documentation to simplify permission requests for app developers.

Multiple system extensions have been suggested to enhance the permission system: The seminal *Kirin* [17] OS extension used combinations of permissions requested by an app to detect potential misuse of permissions and also revealed confused deputy apps on AOSP. *Apex* [55] introduced dynamic and conditional permission enforcement to Android. *TaintDroid* [54] used dynamic taint tracking to reveal for the first time how apps actually use permission-protected data and uncovered a number of questionable privacy practices that motivated enhancements to the permission system and access control on ICC. *Sorbet* [21] was first to model Android permissions and uncovered problems with desired security properties (like controlled delegation of privileges) on Android.

Some system extensions specifically aim at mitigating confused deputies: *XManDroid* [57] primarily augments the permission enforcement with policy-driven access control, where policies specify confused deputy and collusion attacks [18], [98] states. *QUIRE* [56] establishes provenance information along ICC call paths, enabling callees to evaluate their trust in the caller. *IPC inspection* [20] reduces the privileges of callees to the privileges of the caller.

WHYPER [61] and *AutoCog* [62] apply NLP techniques to automatically derive the required permissions from app descriptions, taking developers out of the loop, and check whether described functionality and actually requested permissions correspond. *DescribeMe* [63] takes the opposite track and generates security-centric app descriptions from analysis of app code in order to increase user understanding of the app.

3) Actors' Roles: Platform developers (**A1.●**) and market operators (**A3.●**) are fully responsible for the permission comprehension problems, as the platform enforces use of the current permission system, and the platforms' and the markets' installers communicate the privacy risks of installing applications to users. Library providers (**A4.◐**) contribute to this problem through their permission requests. App developers (**A5.◐**) tend to over-privilege their apps (either for their own needs, or on behalf of library providers their apps use), making apps appear unnecessarily dangerous. End users (**A8.◐**) tend to pay little attention to permissions [9], and only have the option of accepting everything or not installing the app at all.[3] Thus, while end users' behavior eventually opens the door to misuse by malware, end users have limited options and capabilities to detect whether permissions are being misused.

This problem could potentially be fixed by platform developers (**R1.●**) by changing their access control paradigm and avoiding conditions for some of the identified vulnerabilities (e.g. failing to implement fail-safe defaults). Additionally, by helping app developers (**R5.◐**) and library providers (**R4.◐**) in realizing security best practices for defensive programming through tool support [13], [58] (**R7.◐**), this indirectly helps end users. App markets (**R3.●**) could make their permission dialogs more comprehensive, demand justification from app developers and run static analyses on received app packages to adjust permissions accordingly.

4) Lesson Learned: In conventional ecosystems, neither developers nor users were involved in the process of requesting or granting fine-grained permissions to access resources on a computer. Allowing developers to request and define fine-grained permissions and presenting end users permission dialogs is a good idea in theory. However, research illustrates that this approach overburdens both: Developers tend not to focus their efforts on the selection process for permis-

[2] The literature has yet to agree on a fixed term. Other works designate this attack category as permission re-delegation [20], as component hijacking [22], or as capability leaks [24], [92]. We use the term confused deputy [93].

[3] While this has changed with Android v6.0, developers nullify this change by making their apps compatible with older Android versions.

sions [13], while end users neither understand nor pay much attention to Android's permission dialogs [9], [99]. Research has strived to improve permission dialogs [59], [64], [97], but none of these approaches has solved the two-sided usability and comprehension problem. Permission dialogs have issues similar to warning messages: They fail to lead to the desired effect, as users tend to click through them, misunderstand their purpose, and hence do not benefit from them.

Instead of continuing the current line of research, we propose a clean break and a shift towards taking both users and developers out of the loop: Approaches that try to automatically derive the required permissions for an app based on its category, description, and similarity to other apps seem to take a more promising track [61], [62], [63]. Another promising alternative seems to be authorizing entire information flows instead of only access to resources. Although not new [100], [101], this idea seems worth being re-investigated for appified platforms that put the burden of granting permissions onto their end users.

Our assessment (Permission Evolution): The decision to realize permissions as implemented by Android was understandable at Android's launch, but the concept has failed in practice, and was presumably doomed to fail from the beginning.

B. Permission Revolution

A dedicated line of research has investigated the possibilities of extending alternative access-control models to the Android platform to establish more flexible, fine-grained, and mandatory control over system resources and ICC. This research followed two major directions: OS extensions and Inlined Reference Monitoring (IRM).

1) Challenges:

a) Missing Privilege Separation: The most common third-party code distributed with apps is analytic and advertisement libraries that display ads in order to monetize the app (cf. Section III-A) [29]. More than 100 unique ad libraries are available for the different ad networks included in more than half of all apps [25], [26], [27], [28], [29].

The host app and third-party libraries engage in a symbiotic relationship that currently requires mutual trust. Libraries execute in the context of their host app's sandbox and inherit all privileges of their host app. However, ad libraries tend to exploit these privileges and exhibit a variety of dangerous behaviors, including misconduct such as insecure loading of code from web sources [30] as well as collecting users' private information [26]. Inversely, developers of host apps have a strong interest in monetizing their apps. Fraudulent app developers can exploit the symbiotic relationship [31] to surreptitiously steal money from the ad network by faking click events [31]. Android's design failed to provide privilege separation between these two principals [91], worsening the privacy threat of ad libraries to users' data in comparison to conventional browser-based ads [28].

b) Ineffective Security Apps: Android follows the mantra that *"all applications are created equal"* [102]. However, this also implies that apps by external security vendors, such as anti-virus apps, do not have higher privileges than other apps. Studies have investigated to what extent this philosophy influences the efficacy of such security apps [2], [32], [33]. Prior systematization of existing Android malware has evaluated the effectiveness of existing anti-virus apps for Android and reported that detection rates vary from 54.7%-79.6% [2], [33], [32]. One study [33] suggests that platform support for anti-virus apps is essential to improve their efficacy.

c) Lack of Support for Mandatory Access Control: Mobile devices are often used in fields with strong security requirements, such as enterprises and government sectors. Conventional operating systems in those contexts apply advanced access control models that protect more sensitive information (e.g., non-interference between two distinct security levels). The support for mandatory access control is a cornerstone of the platform security of such established systems. Conversely, Android lacks any support for mandatory access control.

While the requirement of supporting advanced access control schemes is intuitive and plausibe, we are not aware of any academic security requirements analysis that focuses on those particular stakeholders (i.e., enterprise and government sectors) on mobile devices and that could describe the particular challenges that come with enabling support for such access control schemes on mobile devices. Only governmental guidelines have been published, e.g., by NIST [103]. Consequently, academic research has explored the particular challenges of adding mandatory and alternative access control models to Android from different angles, not all of which directly relate to high-security deployment.

2) Countermeasures:

To provide advanced access control models and robust defenses against malware on Android, research has followed two main directions for adding access control to Android based on the responsible deployment actor.

a) Alternative Access Control Models: Early work [65], [66], [67] explored how access control within Android's application framework can be more semantically rich and dynamic and introduced mechanisms that have since been adopted by several follow-up works. The seminal *Saint* [65] architecture allows app developers to define policy-based restrictions and conditions on ICC to and from their app. *CRePE* [66] extended Android with context-related access control for system resources, where context is defined as the device state and senseable environment. *TISSA* [67] introduced access control mechanisms for fine-grained data sharing, such as returning filtered, fake, or empty data from calls to framework APIs.

More recently, the *SE Android* [68] project solved the technically complex challenge of porting SELinux-based mandatory access control from the desktop domain to Android. While SE Android focused on the Android OS, *FlaskDroid* [70] demonstrated how SELinux' type enforcement can be extended into the userspace components of the Android application framework and benefit privacy protection.

Prior work specifically addressed the lack of privilege separation between the different security principals on Android. *AdDroid* [27] and *AdSplit* [73] both propose separating advertisement code into separate processes. *LayerCake* [74] investigated the more general problem of secure cross-application interface embedding on Android, e.g., integrating ad libraries or social network plugins into the host app's UI while mitigating common threats such as click fraud, overlays, or focus stealing. *Compac* [72] demonstrates the applicability of stack inspection in conjunction with ICC tagging to establish per-component access control for Android apps.

b) Inlined Reference Monitoring: A parallel line of work [75], [76], [77], [78] has investigated inline reference monitoring [104] for enforcing more fine-grained and dynamic access control policies for privacy protection. These works were mainly motivated by the deployability benefits of binary rewriting as a foundation for IRM in contrast to OS modifications, which empower end users to enhance their privacy independently from platform developers and device vendors.

IRM solutions on Android currently make the inherent tradeoff of abandoning a strong security boundary between untrusted code and reference monitor, and hence their attacker model focuses on curious-but-benign applications rather than on malicious code. Moreover, modifying third-party code involves legal considerations. Most recent advances in this field [79] introduce application virtualization techniques to Android to avoid third-party code modifications and separate the reference monitor from untrusted code.

3) Actors' Roles: The platform developers are able (**A1.●**, **R1.●** and **R2.●**) to integrate more advanced access control models, to offer better privilege separation between third party security principals, and to provide means to integrate external security apps. The lack of support for third-party security apps is particularly noticeable for the platform developer actor, since Android's security philosophy shifts responsibility for privacy protection to end users by forcing them to grant/deny permission requests and by allowing them to load applications from arbitrary sources (i.e. to bypass controlled distribution channels like markets). Furthermore, the problem of missing privilege separation could also be alleviated by ad network providers (**A4.●**, **R4.◐**) by refraining from clearly unacceptable behavior and by implementing security best practices.

Binary rewriting solutions for IRM currently need to be deployed by end users (**R8.●**), who also need to configure policies. Their technical approach would also allow software distribution channels or toolchain providers to implement IRM solutions for apps they distribute/create (**R2.●** and **R7.●**).

4) Lesson Learned: Android adopted design principles from earlier high-assurance systems, and research has proposed valuable access control extensions to their implementations on Android. Although most of the proposed OS extensions are not based on a concrete requirements analysis but rather on postulated challenges, the recent developments of Google's AOSP have a posteriori validated this research; and, in fact, research results can be found in current real-world deployments within the bounds imposed by Google's business model (for instance, SELinux MAC & KNOX [105], dynamic permissions, AppOps, VPN apps, after-market ROMs). Research ideas for privilege separation within app sandboxes, in contrast, should be pushed to maturity and have to be brought to the attention of platform developers. Like mash websites that combine various security principals that are now privilege separated by the browser's sandboxing mechanims, mobile apps that mash various security principals require an adequate privilege separation. IRM solutions are an interim idea, but do not take the user out of the loop (see Section IV-B) and are limited in their security guarantees.

Since access control enforcement on Android has been well studied, the research community should shift focus to the canonical challenges of policy generation and verification. Almost no attention has been given to developing useful and real policies. Drawing from experience on desktop systems, policies are moving targets that require decades to develop; research for mobile systems should support this process. In particular, Android's strong requirement for sharing functionality between apps and the shift to privacy protection are unexplored for global policies. Moreover, at the moment enforcement mechanisms on Android are implemented as best-effort, and the history of OS security has shown the need for verifying complex enforcement mechanisms and their policies.

Our assessment (Permission Revolution): Retrofitting Android with mandatory access control has created valuable ideas that influenced real-world deployments. Better privilege separation of apps should be pushed to maturity. The research community should now refocus on open challenges for policy generation and system verification.

C. Webification

An ongoing trend for mobile apps is *webification*, the integration of web content into mobile apps through technologies like WebView. Seamless integration of apps with HTML and JavaScript content provides portability advantages for app developers. Through its APIs, WebView allows apps a rich, two-way interaction with the hosted web content: Apps can invoke JavaScript within the web page, and also monitor and intercept events in the page as well as register interfaces that web content can invoke to use app-local content outside the WebView sandbox. By now, *mobile web apps* make up 85% of the free apps on Play [39], [37].

1) Challenges: The webification of apps raises new security challenges that are unique to appified mobile platforms.

Foremost, the two-way interaction between a host app and its embedded web content requires app developers to relax the WebView sandboxing. This enables app-to-web and web-to-app attacks [39], [37], [34]. In app-to-web attacks, malicious apps can inject JavaScript into hosted WebViews to extract sensitive user information and use the WebView APIs to navigate the WebView to untrusted websites. In web-to-app attacks, untrusted web content (possibly also injected into an insecure HTTP/S connection [39]) can leverage the JavaScript bridge to the host app to escalate its privileges to the level of its hosting app's process to access local system resources. In particular, popular web app creator frameworks, such as PhoneGap, open a large attack surface for those kind of attacks through their large web-to-app and app-to-web interfaces. [34]

Further,it has been shown [35], [39] that data flows between apps that host different web origins can cross domains through the default Android ICC channels, enabling cross-site scripting and request forgery attacks by malicious apps or untrusted web content within an app. Specifically on mobile platforms, various means enable code to be injected into web content and cross-site scripting attacks to be conducted [38].

2) Countermeasures: To solve the new security challenges raised by webification, different defense strategies have been proposed: *NoFrak* [34] extends the PhoneGap framework with capability-based access control for web origins to restrict access by web content to the functionality of the JavaScript bridge. Along the same lines, *NoInjection* [38] adds sanitization to the bridge of PhoneGap to prevent code injections. *Morbs* [35] proposes an extension to the Android application framework to attach origin information on ICC channels that can cross origin between apps, thus enabling apps to apply a same-origin policy and prevent the reported cross site scripting and request forgery attacks. Additionally, different modifications to the Android WebView and Android IDEs have been discussed [35], [39], such as supporting whitelisting of web origins that have access to the JavaScript bridge; displaying the security of WebView connections to the end user; or lint tools to warn app developers about insecure TLS certificate validation in WebViews.

3) Actors' Roles: Fundamentally, platform developers are required to integrate better isolation of web origins in WebViews and support origin-based access control on data flows (**R1.●**). Additionally, providers of web app frameworks and app publishers are responsible for securing their web-to-app and app-to-web bridges (**R4.●** and **R6.●**).

4) Lesson Learned: The trend towards web apps and usage of web technologies lowered the hurdle for writing apps even more. However, some of the same mistakes known from web applications in browsers were replicated and new problems arose. Cross-origin and web-to-app/app-to-web vulnerabilities constitute serious security challenges for the move towards web apps. However, since such issues are fixable by platform developers and do not require tens of thousands of developers or millions of end users to adopt new security mechanisms, we think this trend is worth pushing in the future.

Our assessment (Webification): Using standard web technology for building apps has proven satisfactory, if somewhat initially shaky. After well-known web security issues have been fixed and integrated with the platform's app sandboxing, this trend should continue.

D. Programming-induced Leaks

This section deals with challenges and countermeasures regarding data leaks caused by developer errors for apps, frameworks, and libraries.

1) Challenges: Android provides a comprehensive set of APIs for app developers. A fraction of these APIs are security-related and provide interfaces for Android's permission system, secure network protocols and cryptographic primitives. Prior work has investigated the quality of security-related API implementations: Fahl et al. [40] investigated security issues with customized TLS certificate validation implementations in Android apps and found widespread, serious problems with how developers used TLS. In follow-up work, they conducted developer interviews to learn the root causes of misusing Android's integrated TLS API and found that the current API is too complex for many developers [42]. Although Android provides safe defaults, in ≈95% of the cases app developers decided to implement customized certificate validation mechanisms, the result being an active MITMA vulnerability.

An analysis on the programming quality of cryptographic primitives such as block ciphers and message authentication codes in Android apps by Egele et al. [43] found that 88% of the analyzed apps made at least one mistake when using those primitives. The authors came to the conclusion that Android's default configuration for cryptographic primitives is not safe enough and that the API documentation in this area is poor. It was also found that apps load code via insecure channels (e.g., http) without verification of the loaded code [30]. Of the hereby analyzed apps, 9.25% are vulnerable to insecure code loading, meaning attackers can inject malicious code into benign apps and turn them into malware. The authors came to the conclusion that this is an API issue, since Android's API does not provide secure remote code loading.

2) Countermeasures: MalloDroid [40] is a static analysis tool to detect broken TLS certificate validation implementations in Android apps. Fahl et al. [42] propose a redesign of Android's middleware/SDK to prevent developers from willfully or accidentally breaking TLS certificate validation. *SMV-Hunter* [41] is a similar approach, additionally applying dynamic code analysis techniques. *CryptoLint* [43] is a static analysis tool to detect misuse of cryptographic APIs on Android. *CHEX* [22] is a static analysis tool to automatically detect component hijacking vulnerabilities. *ScanDroid* [80] is a modular data flow analysis tool for apps, which tracks data flows through and across components. *AndroidLeaks* [81] is a large-scale analysis tool to detect privacy leaks in apps with the intention to reduce the overhead for manual security audits. *FlowDroid* [82] applies static taint analysis techniques to detect (un-)intentional privacy leaks in Android apps.

3) Actors' Roles: Apps that misuse the above security related APIs leave their apps vulnerable to other apps installed on the device (**C2.●**), to malicious dynamic code loading (**C5.●**) or network attacks (**C6.●**).

A common conclusion of the above API misuse studies is that Android's API design does not provide safe defaults (**A1.◐**) in many cases [43]) and when it does, these defaults often do not match the average developer's needs [42] (**A4.●**, **A5.●**). A study to identify the root causes of these issues conducted with Android developers [42] suggests a redesign of existing security related APIs with the developer's needs in mind (**R1.◐**). Better toolchain support to support secure API usage (**R7.◐**) could help the developers of apps (**R5.●**) and library providers (**R4.●**) to write more secure code. App markets (**R3.◐**) could run analyses on apps to prevent insecure apps from being installed on end users' devices.

4) Lesson Learned: Previous research uncovered numerous programming issues. A high number of (new) developers code (mobile/web) apps, and security APIs seem to pose a severe challenge for many of them. Developer interviews illustrated that many inexperienced developers write (mobile/web) apps and struggle to provide the basic functionality, which leaves no room for security and privacy considerations. Many of the provided security APIs allow for very detailed configurations, which seem to overwhelm the average developer and result in insecure/improper selection of security options. Developers are on the front line of the security battle and many of them are currently overburdened. However, user studies with developers [42] illustrate that platform developers could modify the current API design to achieve better security by making APIs more developer-friendly. We argue that it should become common practice to use developer studies to test and improve security and privacy APIs.

Our assessment (Programming-induced Leaks): Existing work on redesigning and simplifying usage of APIs and security-related tools should be extended and complemented by research on currently unexplored areas of developer usability.

E. Software Distribution

Software distribution in the appified world has changed from a decentralized to a centralized model.

1) Challenges: Android's ecosystem has piqued the interest into investigating the impact of its software distribution channels for the protection of end users against malicious apps. A second challenge is the protection of app developers against common problems such as piracy.

a) App Piracy and Malware Incentives: Pioneering work investigated the incentives of malware developers and the state of malware for modern smartphone operating systems like iOS and Android [46]. The authors discovered that the most common malware activities were collecting user information and sending premium-rate SMS messages. This work predicted that in the future, with proliferation of the app markets and advertisement networks for mobile platforms, ad fraud will be a major incentive for malware authors. This prediction has been proven accurate by different follow-up studies [47], [32], [48], [49], [50], [29]. With the exception of a dedicated malware detection analysis [32], these studies focused on *re-packaged* (also noted as *cloned* [49], [50] or *piggybacked* [48]) apps, which have been identified as a major malware distribution method. The common bottom-line of all works (except one [32]) is that markets contain a noticeable number of re-packaged apps. Although all studies found trojan-like malware in the markets, the vast majority of re-packaged apps have been modified to siphon ad revenue from the original app authors (e.g., by exchanging the ad lib or ad identifier), thus suggesting that plagiarists of apps are fiscally motivated. Hence, this majority of re-packaged apps is not strictly malware in the sense that they harm the end user, but instead financially harm the affected app developers [50].

The implication of this research is that besides the known open challenge of protecting end users from malware distributed over markets, another pressing issue is the protection of app developers against plagiarism. Both are important factors in maintaining a healthy appified ecosystem, which needs to be achieved primarily by app markets. A particular challenge towards this goal is that plagiarism not only occurs within a market, but also across markets. To fight plagiarism, some alternative markets like Amazon's App Store require the app developers to participate in their DRM solutions—with limited success [106]. Moreover, the technical enabler for re-packaging apps has to be considered: Android apps are signed by their developers and the signature is used to verify install-time integrity of the installation package and to implement a same-origin update policy. Thus, app developer certificates can be (and are, by default) self-signed certificates whose signature of app packages can be simply replaced with a new signature. This allows re-packaging of apps with a low technical knowledge and effort.

b) Application Signing Issues: Recent work [7] brought up the central role of app markets in appified ecosystems as a new threat for their users. Due to their central role and power when distributing apps, app markets have enormous potential to cheat on their users by withholding apps or updates. A central security mechanism for software distribution is the prior mentioned app signing with self-signed certificates. Investigations [11], [7], [84] illustrate that the way app developers and publishers handle the current app signing mechanism undermines the mechanism's intention: Many developers and publishers use one single key to sign up to 25,000 apps. Without having effective revocation mechanisms at hand, such practices are a serious threat to Android users. For instance, Android allows developers to define permissions that are only available to apps with the same origin (i.e., signing key) in order to establish secure ICC. This same-origin assumption (and with it secure ICC) is defeated by these inappropriate app certification practices.

2) Countermeasures: Different market-enabled solutions have been proposed to address the malware problem: *Meteor* [83] addresses security issues arising from multi-market environments by providing the same security semantics as for single-markets (e.g. kill switches and developer name consistency). *MAST* [84] ranks apps based on their attributes and helps targeting scarce malware analysis resources to apps with highest potential of being malicious. *Application Transparency* [7] addresses Android's application signing issues. It introduces different kinds of cryptographic proofs that allow users to verify the authenticity of apps offered on app markets.

Naturally, different analysis methods evolved to identify malware: *DNADroid* [49] is an approach to detect pirated apps in markets by applying program dependency graphs for methods in candidate apps. *RiskRanker* [85] proposes a proactive zero-day malware detection. *CHABADA* [86] takes a different approach from the prior malware detection tools by relying on anomaly detection: by grouping apps from same categories (e.g., games) by their protected API usage patterns, malicious apps stick out as outliers from those sets.

Ernst et al. [87] divert from the adversarial trust assumptions between app vendor and market operator in prior works by relying on a collaborative verification. Assuming that benign developers will co-operate by annotating their code such that it can be effectively verified, while malicious apps can be reliably rejected, this could enable high-assurance app stores.

AppInk [89] aims at deterring app repackaging through dynamic watermarking of apps. Through an IDE extension, app developers can encode watermarks as triggerable code in their app that can be checked dynamically by a companion app to confirm authorship.

3) Actors' Roles: Platform developers (**A1.●**, **R1.●**) are responsible for fixing key signing issues and allowing for secure distribution of apps in the ecosystem, for instance, distribution of encrypted application packages and full support for PKIs. Additionally, end users (**R8.◐**) could run malware detection software on their devices. However, this would require more effective support for malware detection from the platform developers (see Section IV-B).

App markets (**A3.◐**, **R3.◐**) with their central role in the software distribution process have an enormous impact on security. To prove their correct operations, they can add accountability features [7]. However, also app developers (**A5.●**, **R5.●**) and publishers (**A6.●**, **R6.●**) bear full responsibility for misusing app signing recommendations and have the potential to fix these issues in the future.

4) Lesson Learned: Appification has created an interesting paradigm shift here. Software distribution and installation have become highly centralized. Users typically go to a single app market to search for and install their apps. With their central role in the appified ecosystem, app markets' impact on overall security is enormous. They serve as a line of defense in the fight against malware and could also implement one or more of the many proposed app vetting technologies to protect their users against buggy apps. On the other hand, app markets can also serve as powerful attackers against their own users. They can act as malware distributors or withhold apps or updates.

Although app markets are in a very powerful position, not many of the security and privacy mechanisms proposed by researchers have been adopted by app markets as of today. However, when it comes to privacy, it is potentially not in the best interest of an app market to protect its users. App markets' major motive is monetization by selling apps to their users. As was shown in our systematization, particularly the solutions proposed by researchers to improve users' awareness and control of privacy issues often would require the app markets' cooperation. However, less installs and less lucrative advertising potential could potentially harm app markets' interests. Thus, one result of our work is that researchers should look for additional actors in the ecosystem that could assist in improving users privacy. In particular, app publishers and generators as a strongly emerging pattern for software distribution [7] have not yet received any attention, although their influence on the ecosystem can be considerable. It is unclear to which extent publishers and app generators are trustworthy or are harming the security of apps (e.g., following security best practices) and the privacy of users (e.g., adding tracking code).

Our assessment (Software Distribution): Centralizing software distribution has proven successful for protecting end users against malicious software and for fighting piracy, and should be retained. The threat of malicious app markets is manageable, with countermeasures (almost) ready to be deployed for market-scale application sets. Trustworthiness of app publishers and generators as emerging actors has to be evaluated and established.

F. Vendor Customization/Fragmentation

Fragmentation in appified ecosystems is a wide spread phenomenon since many hardware and software vendors compete for the customer base in the ecosystem.

1) Challenges: The Android ecosystem is fragmented at two different levels: First, Android devices are shipped with different OS versions customized by different vendors. Second, vendors ship their devices with custom system apps. Different works investigated the impact of vendor customizations on the permission enforcement on Android [24], [15], [51], [52], [53] that led to a large number of overprivileged system apps [15]. Moreover, vendor customization significantly increases the phone's attack surface. Vendors introduce higher-privileged apps that act as confused deputies [52] or misconfigurations at framework layer [53], both of which allow unprivileged apps access to protected functionality. Recently, the impact of vendor customizations of the device drivers [107] has been investigated and the study reports very similar results: customizations of Android to fit the vendor-specific hardware have significantly increased the attack surface of the platform and provided attackers access to highly sensible functionality.

2) Countermeasures: As of today no research has been conducted to investigate countermeasures to challenges that stem from fragmented appified ecosystems.

3) Actors' Roles: Vendor customizations, and thus device vendors, are responsible for the security degradations caused by fragmentation and customization (**A2.●**, **R2.●**).

4) Lesson Learned: Android's open ecosystem, in contrast to tighter controlled ecosystems like Apple's iOS, allows vendor customization and fosters the fragmentation that comes along with such customizations. Hence, Android's ecosystem illustrates the potential security risks that such an open approach can induce and should be a warning to concurrent or future appified platforms.

Another lesson to be learned from Android is encouraging vendors to use (system) apps instead of OS patches to provide custom hardware support and force Android to become more modular. Forcing vendors to patch the OS was mainly driven by having only two different privilege levels for apps: system and third party. Eliminating the need for OS patches and allowing vendors to define more privilege levels to integrate customization purely at user space level could reduce fragmentation and drastically reduce the attack surface caused by OS modifications. Although prior works found that vendor app developers make the same mistakes as third-party developers, e.g., over-requesting permissions, bugs in more privileged vendor apps could be more efficiently fixed via the standard app update mechanism in contrast to OS updates. Since vendor app and third party app developers presumably make the same classes of errors, efforts to fix those error classes could be focused instead of having to fight two challenges—apps and OS patches.

Our assessment (Vendor Customization/Fragmentation): Allowing different vendors to customize their devices fueled the adoption process of Android as an appification platform. However, customizing the OS core raised new challenges for platform developers and device vendors. Hence, future fragmentation should focus on system apps rather than OS patches.

G. Software Update Mechanism

Due to centralization of software distribution, app updates are straight forward and can be pushed to millions of users simultaneously. However, fragmentation of the ecosystem makes OS updates very challenging.

1) Challenges: Application life-cycles are very fast paced and updates for actively maintained apps are published in high frequency to markets [29] from where automated update mechanisms distribute them to end users. This is even pushed forward with centralizing updates of security critical libraries such as WebView. In contrast, the situation at OS and application framework level is rather bleak. Thomas et al. [108] present a field study of 20,400 Android devices to measure the prevalence of Android platform specific bugs in the wild. They define a metric to rank the performance of device manufacturers and network operators, based on their provision of updates and exposure to critical vulnerabilities. Their central finding is a significant variability in the timely delivery of security updates across different device manufacturers and network operators, since at least 87% of all investigated devices were vulnerable to at least 11 different vulnerabilities.

In addition, the complexity of upgrading the Android OS version induced problems in the permission management across OS versions [51]. This attack class is currently unique to permission-based mobile systems, such as Android, since the attacker does not corrupt the current system or update image, but instead strategically requests permissions and attributes that are available on the *future* OS version.

2) Countermeasures: No research has thus far investigated countermeasures for challenges that stem from software update mechanisms as implemented on Android. Apart from research, Google has with their latest Android versions changed their update strategy for their Nexus devices [109], [110]. It remains to be seen if other vendors adopt this strategy. Moreover, the *SecUp* [51] app can detect apps that exploit the above mentioned privilege-escalation attack through OS updates.

3) Actors' Roles: Providing OS updates is responsibility of device vendors (**A2**.●, **R2**.●). Platform developers (**A1**.●, **R1**.●) are responsible for introducing the upgrade privilege escalation attack.

4) Lesson Learned: Many researchers expect the platform developer to implement their countermeasures. However, even if that should happen—which is rare, as of today devices are not long-term and frequently maintained by vendors except Google—this expectation is causing slow adaption of new mechanisms and contributes to the fragmentation of the ecosystem [5], [108]. This also opens a large window of opportunity for attackers to compromise the system. Interestingly, appified platforms like Android already have a modularization of software at the application layer. This is inspired by classical high-assurance systems like EROS [111] and in fact, the Binder IPC of Android establishes something like a microkernel-like concept on top of the Linux kernel in userspace. We would like to see this modularization extended to allow modular updates of the system so that security updates can be deployed faster to the end user without requiring a full system update. This is an area where appified platforms are way behind traditional operating systems.

Our assessment (Software Update Mechanism): Since most proposed countermeasures rely on OS updates, and OS fragmentation make these very cumbersome, the platform developers should create better update mechanisms, so that security fixes and countermeasures can be more easily deployed.

V. Conclusion

The central conclusion we draw from this systematization is that, like many new technologies, Android is a story of both victory and defeat. New security mechanisms were introduced without a clear understanding of how these applications would be developed and used, and well-established security mechanisms were re-used to meet the expected security needs of the new general purpose computing platform. Some of the these techniques were a great success, while others failed almost entirely. We draw the following meta-conclusion:

Our meta-assessment: Some aspects worked out beautifully, e.g., centralizing software distribution helps to tackle critical security issues and makes fighting piracy and malware easier. Other approaches had initial difficulties, but are now more or less on track after research has helped to identify and bridge them. Examples comprise easier-to-use APIs that have started to replace hard-to-use but well-intended security APIs over the last few years, as well as the concept of Webification that has enabled more developers to produce their own apps. However, some approaches should be re-thought from the beginning and arguably abandoned for designs of future OSes: Permission dialogs for end users should be removed entirely, since they failed for the same reasons warning messages have failed since the dawn of computing.

Acknowledgements

This work was supported by the German Federal Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA) via the projects BASE (#16KIS0344) and VFIT (#16KIS0345).

References

[1] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing Android: A survey, taxonomy, and challenges," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 58:1–58:45, May 2015.

[2] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. 33rd IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 2012.

[3] P. McDaniel and W. Enck, "Not so great expectations: Why application markets haven't failed security," *Security Privacy, IEEE*, vol. 8, no. 5, pp. 76–78, Sept 2010.

[4] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[5] Open Signal, "Android fragmentation visualized (august 2015)," http://opensignal.com/reports/2015/08/android-fragmentation/, last visited: 11/06/2015.

[6] Samsung, "Knox," Online: https://www.samsungknox.com, last visited: 11/13/2015.

[7] S. Fahl, S. Dechand, H. Perl, F. Fischer, J. Smrcek, and M. Smith, "Hey, NSA: Stay Away from my Market! Future Proofing App Markets against Powerful Attackers," in *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[8] Google, "Google Report: Android Security 2014 Year in Review," 2014.

[9] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. ACM, 2012.

[10] A. P. Felt, S. Egelman, and D. Wagner, "I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns," in *Proc. 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'12)*. ACM, 2012.

[11] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot, "Understanding and improving app installation security mechanisms through empirical analysis of android," in *Proc. 2nd Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*. ACM, 2012.

[12] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission evolution in the Android ecosystem," in *Proc. 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 2012.

[13] A. Porter Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conference on Computer and Communication Security (CCS'11)*. ACM, 2011.

[14] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[15] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[16] D. Barrera, H. G. Kayacik, P. C. Van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proc. 17th ACM Conference on Computer and Communication Security (CCS'10)*. ACM, 2010.

[17] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conference on Computer and Communication Security (CCS'09)*. ACM, 2009.

[18] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proc. 13th Information Security Conference (ISC'10)*. Springer-Verlag, 2010.

[19] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*. ACM, 2011.

[20] A. Porter Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[21] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing android's permission system," in *Proc. 17th European Symposium on Research in Computer Security (ESORICS'12)*. Springer-Verlag, 2012.

[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[23] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[24] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[26] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*. ACM, 2012.

[27] P. Pearce, A. Porter Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and advertisers in Android," in *Proc. 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS'12)*. ACM, 2012.

[28] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in android ad libraries," in *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[29] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *Proc. 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. ACM, 2014.

[30] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[31] J. Crussell, R. Stevens, and H. Chen, "MAdFraud: Investigating Ad Fraud in Android Applications," in *Proc. 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, 2014.

[32] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[33] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: evaluating Android anti-malware against transformation attacks," in *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)*. ACM, 2013.

[34] V. S. Martin Georgiev, Suman Jana, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[35] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[36] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *Proc. Information Security Applications*. Springer-Verlag, 2014.

[37] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.

[38] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[39] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Proc. 2015 Mobile Security Technologies Workshop (MoST'15)*. IEEE, 2015.

[40] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory love Android: An analysis of Android SSL (in) security," in *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[41] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[42] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[43] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[44] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, You, Get Off of My Clipboard," in *Proc. 2013 Financial Cryptography and Data Security (FC'13)*. Springer-Verlag, 2013.

[45] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[46] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

[47] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12)*. ACM, 2012.

[48] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proc. 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. ACM, 2013.

[49] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Proc. 17th European Symposium on Research in Computer Security (ESORICS'12)*. Springer-Verlag, 2012.

[50] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "Adrob: Examining the landscape and impact of android application plagiarism," in *Proc. 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*. ACM, 2013.

[51] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *Proc. 35th IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 2014.

[52] A. Moulo, "Android OEM's applications (in)security and backdoors without permission," http://www.quarkslab.com/dl/Android-OEM-applications-insecurity-and-backdoors-without-permission.pdf.

[53] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace, "Hare hunting in the wild android: A study on the threat of hanging attribute references," in *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[54] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th Usenix Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010.

[55] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android permission model and enforcement with user-defined runtime constraints," in *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS'10)*. ACM, 2010.

[56] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[57] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[58] T. Vidas, N. Christin, and L. Cranor, "Curbing Android permission creep," in *Proc. Workshop on Web 2.0 Security and Privacy 2011 (W2SP'11)*. IEEE, 2011.

[59] P. G. Kelley, L. F. Cranor, and N. Sadeh, "Privacy as part of the app decision-making process," in *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'13)*. ACM, 2013.

[60] M. Harbach, M. Hettig, S. Weber, and M. Smith, "Using personal examples to improve risk communication for security and privacy decisions," in *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*. ACM, 2014.

[61] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: towards automating risk assessment of mobile applications," in *Proc. 22nd Usenix Security Symposium (SEC'13)*, 2013.

[62] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[63] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for android apps," in *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[64] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov, "Android permissions remystified: A field study on contextual integrity," in *Proc. 24th USENIX Security Symposium (SEC'15)*. USENIX Association, 2015.

[65] M. Ongtang, S. E. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," in *Proc. 25th Annual Computer Security Applications Conference (ACSAC'09)*. ACM, 2009.

[66] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-related policy enforcement for Android," in *Proc. 13th Information Security Conference (ISC'10)*. Springer, 2010.

[67] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST'11)*. Springer-Verlag, 2011.

[68] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[69] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proc. 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

[70] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies," in *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[71] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "Asm: A programmable interface for extending Android security," in *Proc. 23rd USENIX Security Symposium (SEC'14)*. USENIX Association, 2014.

[72] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in android," in *Proc. 4th ACM Conference on Data and Application Security and Privacy (CODASPY'14)*. ACM, 2014.

[73] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications," in *Proc. 21st Usenix Security Symposium (SEC'12)*. USENIX Association, 2012.

[74] F. Roesner and T. Kohno, "Securing embedded user interfaces: Android and beyond," in *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[75] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for android applications," in *Proc. 21st Usenix Security Symposium (SEC'12)*. USENIX Association, 2012.

[76] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: fine-grained permissions in android applications," in *Proc. 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'12)*. ACM, 2012.

[77] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," in *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[78] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard – enforcing user requirements on Android apps," in *Proc. 19th International Conference on Tools and Algorithms for the

Construction and Analysis of Systems (TACAS '13). Springer-Verlag, 2013.

[79] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android," in *Proc. 24th USENIX Security Symposium (SEC'15)*. USENIX Association, 2015.

[80] A. Chaudhuri, A. Fuchs, and J. Foster, "SCanDroid: Automated security certification of Android applications," University of Maryland, Tech. Rep. CS-TR-4991, 2009.

[81] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *Proc. 5th International Conference on Trust and Trustworthy Computing (TRUST'12)*. Springer-Verlag, 2012.

[82] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, 2014.

[83] D. Barrera, W. Enck, and P. C. V. Oorschot, "Meteor: Seeding a security-enhancing infrastructure for multi-market application ecosystems," in *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[84] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: Triage for market-scale mobile malware analysis," in *Proc. 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'13)*. ACM, 2013.

[85] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in *Proc. 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. ACM, 2012.

[86] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th IEEE International Conference on Software Engineering (ICSE'14)*. ACM, 2014.

[87] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[88] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proc. 24th USENIX Security Symposium (SEC'15)*. USENIX Association, 2015.

[89] W. Zhou, X. Zhang, and X. Jiang, "AppInk: watermarking android apps for repackaging deterrence," in *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)*. ACM, 2013.

[90] T. Wyatt, "Security alert: Geinimi, sophisticated new android trojan found in wild," https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/, 2010, last visited: 11/06/15.

[91] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[92] P. P. Chan, L. C. Hui, and S.-M. Yiu, "Droidchecker: analyzing android applications for capability leak," in *Proc. 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*. ACM, 2012.

[93] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, Oct. 1988.

[94] Google, "Review app permissions thru android 5.9," https://support.google.com/googleplay/answer/6014972?hl=en, last visited: 11/13/205.

[95] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner, "How to ask for permission," in *hotsec12*. USENIX Association, 2012.

[96] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proc. 33rd IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 2012.

[97] B. Liu, J. Lin, and N. Sadeh, "Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help?" in *Proc. 23rd International Conference on World Wide Web (WWW'14)*. ACM, 2014.

[98] C. Marforio, H. Ritzdorf, A. Francillon, and S. Čapkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proc. 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 2012.

[99] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring user confidence in smartphone security and privacy," in *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. ACM, 2012.

[100] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, 1997.

[101] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, 2008.

[102] Open Handset Alliance, "Android," http://www.openhandsetalliance.com/android_overview.html, last visited: 11/13/2015.

[103] M. Souppaya and K. Scarfone, "NIST Special Publication 800-124 Revision 1: Guidelines for Managing the Security of Mobile Devices in the Enterprise," Jun. 2013.

[104] Ú. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Ph.D. dissertation, Cornell University, January 2004.

[105] R. Mallempati, "Google i/o recap, part 1: Google is serious about enterprise mobility," Online: https://www.mobileiron.com/en/smartwork-blog/google-io-recap-part-1-google-serious-about-enterprise-mobility, Jun. 2014, last visited: 11/13/2015.

[106] lohan, "Antilvl: android cracking," http://androidcracking.blogspot.in/p/antilvl_01.html, last visited: 11/06/15.

[107] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *Proc. 2014 IEEE Symposium on Security and Privacy (SP'14)*. IEEE Computer Society, 2014.

[108] D. R. Thomas, A. R. Beresford, and A. Rice, "Security metrics for the android ecosystem," in *Proc. 5th ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'15)*. ACM, 2015.

[109] Google, "Nexus security bulletins," https://source.android.com/security/bulletin/index.html, last visited: 11/13/2015.

[110] R. Brandom, "Android marshmallow's best security measure is a simple date," http://www.theverge.com/2015/9/29/9415313/android-marshmallow-security-update-vulnerability, 2015, last visited: 11/12/2015.

[111] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, 1999.