Bringing Balance to the Force: Dynamic Analysis of the Android Application Framework

Abdallah Dawoud
CISPA Helmholtz Center for Information Security
abdallah.dawoud@cispa.saarland

Sven Bugiel
CISPA Helmholtz Center for Information Security
bugiel@cispa.saarland

***Abstract*—Android's application framework plays a crucial part in protecting users' private data and the system integrity. Consequently, it has been the target of various prior works that analyzed its security policy and enforcement. Those works uncovered different security problems, including incomplete documentation, permission re-delegation within the framework, and inconsistencies in access control. However, all but one of those prior works were based on static code analysis. Thus, their results provide a one-sided view that inherits the limitations and drawbacks of applying static analysis to the vast, complex code base of the application framework. Even more, the performances of different security applications—including malware classification and least-privileged apps—depend on those analysis results, but those applications are currently tarnished by imprecise and incomplete results as a consequence of this imbalanced analysis methodology. To complement and refine this methodology and consequently improve the applications that are dependent on it, we add dynamic analysis of the application framework to the current research landscape and demonstrate the necessity of this move for improving the quality of prior results and advancing the field. Applying our solution, called DYNAMO, to four prominent use-cases from the literature and taking a synoptical view on the results, we verify but also refute and extend the existing results of prior static analysis solutions. From the manual investigation of the root causes of discrepancies between results, we draw new insights and expert knowledge that can be valuable in improving both static and dynamic testing of the application framework.**

I. INTRODUCTION

Android provides apps with rich features, such as location tracking, taking pictures, sensing, or access to managed user data. Those features are offered to app developers through Android's application framework API and are implemented by system services and system apps. The application framework is responsible for controlling access to system resources and user data, and consequently plays a crucial role in protecting system integrity and user privacy. This access control is based on the privileges ("permissions"), identity, or other attributes of calling processes. However, the application framework has continuously grown over the various Android releases and has become a massive code-base that is even further extended and modified by OEMs for their own purposes. As a result, the framework also became less and less transparent, and the correct enforcement of security policies harder to judge. This problem is best exemplified by the seemingly simple task to determine which API of the application framework is guarded by which security conditions (i.e., a *permission mapping*), something that has not been satisfyingly solved until today.

In fact, over the last decade, Android's application framework has been the target of a still ongoing line of research [20], [11], [12], [47], [6], [4], [28], [5] to analyze and model the complex security policy protecting the APIs of system services. The modeled policy has been central to research and industrial applications in assisting developers to write least-privileged apps [51], to detect malware and over-privileged apps [18], [48], [44], [9], [57], [40], [27], to detect vulnerabilities in the application framework itself [47], [56], [6], [4], [28], [29], and other use-cases [13], [35], [43]. This large body of research and its applications underline the significance of a complete and sound modeling of the security policy enforced in Android's application framework. The consequences from lacking such a model include, for instance, developers writing over-privileged apps that contribute to eroding users' trust and comprehension of the permission system [37], [31], app analysis relying on incomplete and inaccurate information to determine the severity of apps' API calls and consequently misclassifying apps or malware [9], unclear consequences of OEM modifications on the default security policy and whether they weaken it [6], or mistakes in the default security policy jeopardizing system's integrity and user's privacy [29].

Looking back at this line of research, we make some observations on the methodology. Although this research started with a dynamic analysis solution by Porter Felt et al. [20], ever since all subsequent solutions [11], [12], [47], [6], [4], [28], [5] have been exclusively relying on static analysis. The switch to static analysis is understandable given that there exist well-established and available static analysis tools (predominantly Wala [3] and Soot [50]), which are capable of achieving high code coverage. However, static analysis also has its own well-known limitations, especially when applied to a massive, complex code base like that of the application framework, including over-approximation, simplification of analysis, and inefficiency in bridging IPC. Given that there is no ground truth for the security policy in Android and there is no systematic approach for verifying results of static analysis at scale, it is hard to judge the accuracy and completeness of those results. In fact, looking at other software testing domains [22], [52], [15], [58], [7], [42], it is common to compensate for inherent limitations of static analysis by combining it with other techniques, such as dynamic testing. Given the recent advances in dynamic analysis techniques for Android, e.g., dynamic code

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23106
www.ndss-symposium.org

instrumentation and the emergence of powerful fuzzing tools, committing to *only* static approaches for analyzing Android's application framework is an unjustified methodology whose results will be tipped to a one-sided view and be unnecessarily bound by inherent limitations.

Drawing inspiration from the software engineering domain [22], [52], [15], [58], [7], [42], we show that combining static and dynamic analysis should be the natural next step in this line of work to analyze Android's pivotal application framework. The concrete challenge at hand is that while there exist well-established static analysis tools [3], [50], no proper solution for *dynamically* testing and modeling the security policy of the application framework exists—the last solution [20] is obsolete and technically as well as conceptually limited (see Section III for a detailed discussion).

Our contributions: *First,* to fill the gap that is caused by the absence of a dynamic testing tool for analyzing the application framework, we introduce in this paper a dynamic testing tool, called DYNAMO, that is designed with two objectives in mind: (1) analyzing the security policy of the application framework for different versions of Android, and (2) revisiting the existing results of static analysis tools. While DYNAMO uses well-known techniques of dynamic testing and fuzzing (e.g., runtime instrumentation, feedback-driven testing), the novelty of this work comes from the right application of those techniques in studying the security policy of the application framework. As such, DYNAMO does not replace current solutions in this domain but complements the current methodology that has been tipped towards static analysis. To enable the community to reproduce and extend our results, we open source our tool and results [16].

Second, we use our tool to reproduce, extend, and verify the results of prior works for building permission mappings [12], [5], [11], for discovering permission re-delegation vulnerabilities [29], and for detecting inconsistencies in the security policy of two Android versions [47]. We further use a permission mapping built by DYNAMO for the latest Android release to assess the correctness and completeness of permission annotations in Android's developer documentation. DYNAMO's results have an immediate security impact (i.e., better developer documentation and discovery of permission inconsistencies and misconfigurations) but also have an implicit impact through more trustworthy permission mappings for a wide range of dependent security applications on Android (e.g., [44], [40], [9], [18], [27], [36], [48], [57], [4], [6], [28], [29], [47], [56], [13], [35], [43], [51]).

Third, our root cause analysis of the discrepancies between existing results and our dynamic analysis results provides valuable feedback to the designers of both static and dynamic testing tools. Concretely, we highlight implementation and approach-inherent shortcomings in existing tools that are based on static analysis. Through a quantitative and qualitative analysis, we measure the negative impact of those shortcomings on previous results. We additionally contribute new insights that help to better understand how system services are interconnected and protected, and we add crucial contextual information about access control enforcement, such as the resolution of the subject's identity and locality and order of security checks (which are fundamental to permission re-delegation and inconsistency analysis [29]).

DYNAMO clearly improves on the status quo. Specifically, our evaluation of the permission mapping by Arcade [5]—the latest and most sophisticated permission mapping—shows that the mapping of 76.1% of 951 analyzed APIs (that exist in both Arcade's and DYNAMO's mapping) are verified, 3.1% are incorrect, and 10.6% are incomplete. DYNAMO additionally reports permission mappings for 343 APIs that are missing from Arcade's results while DYNAMO misses 247 APIs reported by Arcade. We shared our findings with Arcade's authors who, following our discussion, upgraded their tool and published a new permission mapping accordingly. We discuss the changes to their original permission mapping separately in Section 3. We also evaluated results from ARF [29] for discovering permission re-delegations within system services and found that 5 out of 33 reported vulnerabilities are False Positives (FPs) while 10 could be confirmed as true vulnerabilities. Our consistency analysis of the permission enforcement revealed 5 sensitive APIs that are unprotected and 65 APIs with permission misconfigurations. Finally, evaluating Android's developer documentation [24] for 439 APIs showed that it is incomplete (66 APIs) and imprecise (9 APIs), which we reported to Google for corrective actions. Section VI discusses the security impact of these findings.

All in all, our contributions demonstrate the importance of a more diverse set of analysis tools for Android's middleware and the feedback loop between those tools. The fact that DYNAMO reproduces the vast majority of results of previous works, reports new findings, and adds trustworthiness to the results along the way, shows how static and dynamic analysis can go hand-in-hand in providing better results.

II. BACKGROUND

Android is a Linux-based OS that is characterized by its open-source software stack. Most relevant to this work from that stack is the application framework, which offers a variety of system services that expose a wide range of features to app developers and other components of the framework. Those features are implemented as `Service` APIs and they are reachable over an Inter-Process Communication (IPC), called Binder. A subset of those APIs is encapsulated in `Manager` APIs (e.g., by WifiManager and LocationManager classes) that sanitize inputs and invoke service APIs over IPC (e.g., of WifiService and LocationManagerService classes). However, the manager APIs can be circumvented by invoking service APIs directly, e.g., via Java reflection or native code.

Invoking an API is a multi-step operation (see Figure 1) that starts by querying for the reference to the target service from a central directory, called `ServiceManager` that keeps track of all registered system services in the system. This reference is then used to construct a proxy object that exposes the APIs of the target service. When a specific API from the proxy is invoked, the kernel transfers the call's payload to the service side, conveying the Linux UID and PID of the calling process along the way. At the receiving end, the payload is decoded and the target method is invoked.

One of the crucial features of the Binder IPC is conveying the caller's identity (i.e., UID and PID of the calling process) to the callee. This feature is used to implement high-level access

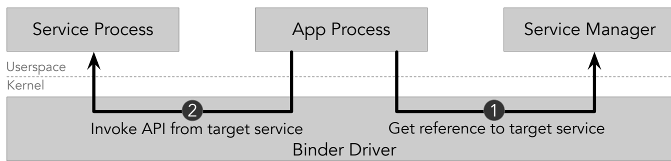

Fig. 1. Multi-Step Binder IPC to call Service API

control protecting sensitive APIs. This access control relies on permissions as the main policy. Permissions are special strings centrally managed by the `PackageManagerService` (`PMS`) and assigned to the apps, e.g., after being granted to the app by the user. Every app is identified by its UID and granted permissions are bound to the UID. The access control further uses three other known types of checks to regulate access to sensitive APIs. Those types are: 1) Checks of the caller's UID and PID that are retrieved via `Binder.getCallingUid/Pid()` and are used to allow calls from specific privileged contexts. For example, they can be used to exclusively allow the system (`getCallingUid()== 1000`) or calls from the same process (`getCallingPid()== currentPid`) to execute a specific API. 2) Across-profile checks that use the caller's `userId` derived from the caller's UID to distinguish between calls from the current and other profiles (where a profile corresponds to a physical user) and treat them differently. For example, some APIs might reject calls from secondary profiles or enforce more permissions on them. This type of checks also includes user restriction checks that are used to restrict calls from work profiles [26] to the main profile. 3) AppOps permissions that are complementary to normal permissions to change the runtime behaviour. For example, when a permission is granted but the AppOps permission is "ignored," some APIs are configured to return a dummy data or to fail silently. Since access control in native system services has not been studied yet, we rely on our own observations that suggest that the access control in native APIs also uses permissions and checks based on UID/PID. For example, permissions checks in native APIs are routed to the `PMS` as a policy decision point via an intermediate service called `PermissionController`.

Finally, the high-level access control is complemented by low-level Discretionary and Mandatory Access Control (DAC and MAC, respectively). DAC uses conventional Unix permissions (based on groups and UIDs) to confine application sandboxes and processes, e.g., from circumventing the service APIs by directly accessing the resources encapsulated by the system services and apps, such as private user data or device drivers. Since Android 5.0, fully enforced SELinux is used for MAC to reinforce the confinement of processes and harden the system against privilege escalations.

III. Related Works and Motivation

We summarize related works that studied Android's application framework and motivate our work.

Permission Mapping: Four major works [20], [11], [12], [5] built permission mappings of system APIs in Android. Stowaway [20] pioneered this research by using dynamic testing. It built on the intuitive idea of invoking system APIs and observing their required permissions. Stowaway ran a testing app on top of a modified build of Android that logged the permissions checked for the testing app. To invoke the APIs and trigger the permission checks, the app executed unit tests that were semi-automatically generated by Randoop [2]. Motivated by the low code coverage of Stowaway's dynamic approach and its considerable manual effort, PScout [11] proposed a static analysis technique using the Soot framework [50] to build permission mappings. It constructed a call graph of the application framework, marking all permission checks (e.g., `checkPermission` method), and performed backward reachability analysis starting from those check points to the APIs that use them. Axplorer [12] revisited the problem but based on the WALA framework [3]. It highlighted new insights (e.g., how to identify entry points) and handled new design patterns (e.g., message passing) that were not covered by PScout, causing imprecision in the produced mapping. Most recently, Arcade [5] used static analysis also based on WALA to add path-sensitivity to the reported mapping (i.e., different execution paths that are controlled by inputs require different sets of permissions). Arcade's mapping considers the relation between different security checks (e.g., whether enforced in conjunction or disjunction). It additionally considered more attributes that influence the access control decision besides the permissions (e.g., caller's UID/PID). Closely related to building permission mappings in Android, Kobold [17] studied the security policy of APIs that are exposed over IPC to third-party app developers in *iOS*. Kobold's approach is similar to Stowaway in its design but is currently more limited by the closed source nature of iOS.

Vulnerability Detection in the Security Policy: Another line of research that studied Android's application framework focused on discovering discrepancies in access control enforcement within system services, e.g., two APIs that are protected by different security conditions but lead to the same functionality or data sink. The first task to enable such analysis is to model the security policy of system APIs. Kratos [47], DiffDroid [6], and AceDroid [4] used a predefined list of authorization checks, e.g., *checkPermission* and *hasUserRestriction*, and incrementally but manually complemented this list to define the security policy of individual APIs. To remove the dependency on the user-defined list of authorization checks, ACMiner [28] introduced a semi-automatic and heuristic-driven approach to build this list. Centaur [41] proposed symbolic execution in conjunction to static analysis to discover and verify the inconsistencies. However, Centaur requires access to the source code and cannot be used for closed-source OEM images. Other works [59], [30] analyzed parameter-sensitive APIs that are improperly protected. Exploiting those APIs would disturb the system's state or escalate the caller's privileges. Closely related to inconsistency detection, ARF [29] employed static analysis and manual code inspection techniques to discover permission re-delegation within Android system services where one API calls another protected API and enforces less restricting permissions compared to the ones that are enforced when directly calling the target API.

Fuzzing for Vulnerability Detection: One of the early works applying fuzzing to Android's application framework is Buzzer [14], which is a black-box fuzzing tool that focused on testing input validation of system APIs using manually crafted inputs. Another similar, but more advanced, work is BinderCracker [21] that fuzzed the system APIs with faulty

payloads causing permission leakage or Denial-of-Service. Ex-Hunter [55] targeted improperly handled exceptions in system services that crash the system when triggered. ASVHunter [33] fuzzed the `system_server` process (i.e., the host process for the majority of system services) and exploited weaknesses in its concurrency control mechanism. Chizpurfle [34] introduced a grey-box fuzzing leveraging runtime code instrumentation to provide feedback on code coverage and refine input generation. FANS [38], the most recent work, fuzzed native services for vulnerability detection. It developed a multi-level discovery technique to access native services hidden in other services.

Problem Statement and Motivation

Studying the works that investigated the security policy of Android's application framework [20], [11], [12], [47], [6], [4], [28], [5], [59], [30] reveals common features. The most common one is that they all modeled the security policy and demonstrated interesting use-cases on top of it, such as vulnerability [29], [30] and inconsistency detection [47], [4], [28] in the policy, privilege escalation detection in apps [20], [5], studying the evolution of Android's permission system [11], and cross-OEM analysis [6], [59]. The second common feature is that they all, apart from Stowaway [20], relied on static analysis techniques for their task.

Unfortunately, static analysis tools, although achieving high code coverage, tend to over-approximate, especially when applied to massive code-bases like the one of the application framework. There are a few reasons for that. For instance, static analysis cannot resolve runtime variables causing uncertainty on which path to follow, thus either following *all* paths or making a hopefully right choice [47], [5]. Additionally, it requires a good understanding of the code under test and the used design patterns to be precise [12]. Another reason is that, depending on the goal of the analysis, static analysis might become expensive and forces the developers to compromise precision for performance [8], [47], [28]. In addition to those fundamental limitations of static analysis, solutions targeting specifically Android's application framework also face the challenges of handling chained IPC between the system services and testing services implemented in native C/C++ code.

While dynamic analysis can compensate for the limitations of static analysis, as has been customary in the software engineering domain, Stowaway [20], the only dynamic solution for testing the application framework, is currently outdated, unavailable, and technically and conceptually limited. Specifically, when Stowaway is deployed as a permission mapper, it only detects the first encountered permission ignoring the permissions that are enforced in conjunction and disjunction (i.e., disregards path-sensitivity), ignores non-permission checks (e.g., hardcoded UIDs), requires source-code modifications, entails considerable manual effort in input generation, achieves low API-coverage, reports false positives as it struggles in isolating noise in its feedback channel, and ignores native code. Stowaway is further challenged by the security improvements added to Android's access control since its publication (e.g., the introduction of runtime permissions in Android 6.0 and SELinux in Android 5.0). As such, Stowaway is technically and conceptually limited to properly analyze the security policy of the application framework. Additionally, although fuzzing tools in Android exist (e.g., [14], [21], [55], [33], [34], [38]), they only work as vulnerability scanners in the framework—a goal that is different from modeling the security policy. In fact, those fuzzers are designed as highly-privileged processes that bypass all access control checks protecting the fuzzed APIs. Identifying the security checks in exploitable paths is usually done manually as a final step in the analysis and is not the focus of those works. Thus, they are also technically and conceptually impractical for our task.

All in all, the results of the prior works that studied the security policy of the application framework have shaped our understanding of how the framework works and clearly advanced the field. However, with the absence of proper dynamic testing tools, the adopted methodology has been tipped to a one-sided view (i.e., static analysis) and bound by its inherent limitations. This raises valid questions, such as: 1) To what extent are the results of this methodology accurate and complete? 2) Can we systematically and automatically verify them? 3) After 10 years of studying Android's security policy, did we actually fully uncover it? and 4) To what extent can solutions of static analysis benefit from complementary dynamic testing. In this work, we revisit the problem of modeling Android's security policy by using dynamic testing. We show that our approach is able to confirm, complete, and refute results from previous works that studied Android's security policy only with static analysis. *Thus, we make in this paper a strong argument for adding dynamic code analysis tools for Android's application framework to the arsenal of security researchers. A synoptical view cannot only help to improve analysis results but also provide feedback to improve the analysis techniques.*

IV. Design and Implementation

We introduce in this section the design and implementation of Dynamo, our dynamic code analysis tool for Android's application framework.

A. Overview

Dynamo builds on the simple idea of invoking the APIs of system services from an unprivileged app and detecting the security checks that protect those APIs. This approach is similar to that of Stowaway [20], but our solution overcomes Stowaway's conceptual and technical limitations and captures the intricate details of the access control in Android's application framework. For instance, Dynamo leverages insights from recent work [5] that highlighted the need for modeling the security policy with respect to path-sensitivity and maintaining the relation between different checks. Dynamo's design is further tuned to detect non-permission security checks (e.g., abstract checks on the UID/PID of the calling process, cross-user checks, AppOps permissions, etc) in the Java layer as well as native code, which is missing in all previous works. To enable Dynamo to operate on Android images from Google and other OEMs, it is designed to exploit runtime instrumentation instead of modifying AOSP's source code. Dynamo also employs techniques from the software testing domain for automatic input generation based on a short list of predefined seeds. The testing itself can be tailored to the specifics of each API (i.e., custom inputs and testing frequency). This feature makes Dynamo's results reproducible and deterministic for

the same setup. Moreover, DYNAMO has proven to efficiently work on all Android versions starting from Android 6.

At the heart of DYNAMO is a bi-directional communication channel that connects DYNAMO with the framework's components and that serves two purposes. First, it provides the means to instrument the framework's runtime behaviour, e.g., changes the behaviour of selected methods. Second, it provides a reliable feedback channel to report the security checks that are triggered as a direct consequence of testing a specific API. The feedback includes the execution traces of the security checks, method-coverage information, and important contextual information, such as the identity that is used in those checks, their locations, and their order. The reported coverage information can be leveraged to enhance the input generation.

B. Research Questions for Dynamic Code Analysis

Several works [14], [21], [34], [38], [55], [33] fuzzed the application framework for vulnerability detection. While those works addressed several technical challenges, dynamically testing the framework for the modeling of the security policy of system APIs brings additional challenges. Stowaway [20] highlighted some of them, such as how to trigger, capture, and report permission checks. Although Stowaway serves as a great source of inspiration to this work, we believe that the problem of dynamically analyzing the security policy of Android's framework in general, and building permission mapping in particular, has evolved since the introduction of Stowaway. We recognize the challenges and formulate them in a series of derivative research questions that shaped DYNAMO's design.

(RQ_1) How to identify the entry points of system APIs and invoke them? The APIs are scattered across different system services and implemented in Java, native code, or both. While all previous works focused on analyzing Java-based APIs, the security policy of native APIs remains ambiguous.

(RQ_2) How to build valid API inputs? While vulnerability scanners invest in generating inputs that trigger bugs or crashes, modeling the security policy requires building syntactically and semantically valid inputs that trigger all security checks protecting the tested API. Automatically building such inputs might unnecessarily exhaust the testing budget if not complemented with a direct feedback of achieved coverage.

(RQ_3) How to measure coverage? We, unfortunately, lack the well-established tools for measuring code coverage in Android and best efforts [34] entail a huge overhead. However, measuring coverage is crucial to refine the input generation strategy and subsequently discover deep-hidden security checks.

(RQ_4) How to detect and report different security checks? While some security checks are centrally managed (e.g., permissions) and are straightforward to report (i.e., by placing necessary hooks into the corresponding checking services), other checks are inlined (e.g., comparing the caller's UID with a predefined privileged UID) and pose a big challenge for dynamic testing to discover them. Additionally, reporting all checks requires a reliable feedback channel that isolates the security checks triggered by testing an API from the noise of unrelated security checks.

(RQ_5) How to construct the feedback channel? The feedback channel can be implemented in the middleware via direct

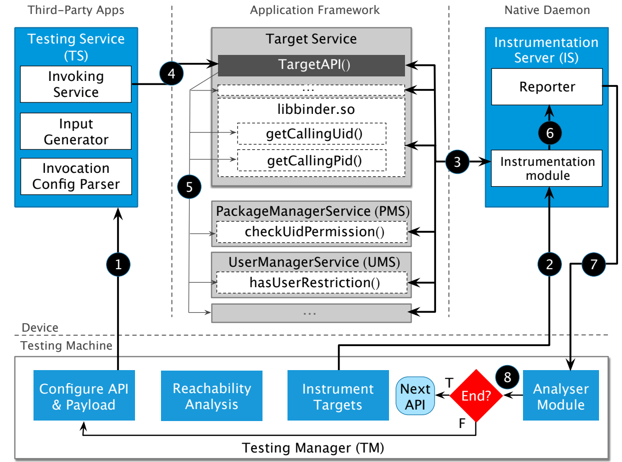

Fig. 2. Steps of one testing iteration by DYNAMO

modifications to the OS. Apart from the overhead associated with this approach, it is not scalable to other closed-source images of different OEMs. Being able to also analyze the security policy of OEMs is crucial to uncovering and mitigating possible erroneous changes on AOSP's default policy.

(RQ_6) How to preserve the relation and order of security checks? Recent works [5], [29] have shown that reporting permissions as a set over-simplifies the modeling of the intricate access control of system APIs and causes imprecision in the applications built on top. Instead, different sets of security checks must be reported for different execution paths where each path is controlled by the API's input or system state (path-sensitivity). Each set must also retain the order of the checks.

C. Implementation of Dynamo

DYNAMO is a grey-box testing solution that consists of two stages of operation to build the security policy of an API. The first stage focuses on testing the API by exercising different input sets with the goal of increasing code coverage. In the second stage, those results are analyzed based on predefined association rules to model the security policy of the target API. Figure 2 depicts an overview of DYNAMO's main components and presents the sequential execution steps of one testing iteration for one API. The `Testing Service` (`TS`) is shipped as an app component that is installed on the tested device. It is responsible for generating inputs, invoking the target API, and reporting the invocation result. The `Instrumentation Server` (`IS`) is a daemon process that is responsible for constructing the feedback channel that reports missing security checks and coverage information. It is additionally responsible for modifying methods' behaviour at runtime at the request of the `Testing Manager` (`TM`). The current version of DYNAMO uses a dynamic instrumentation toolkit, called Frida [46] to run the `IS`. The `TM` runs on the tester's machine and orchestrates the whole process, including setting the seeds for input generation, defining the tested API, identifying the hosting service, triggering the instrumentation of targets, triggering API invocation, collecting the invocation and instrumentation feedback, and finally analyzing the results.

Detecting and Preparing Target Devices: The `TM` uses Android Debugging Bridge (ADB) to discover all connected

devices (be it hardware or emulated) and considers them as targets for analysis. As we will explain later, DYNAMO relies on disabling the anti-reflection mechanism that is deployed in recent versions of Android and of the SELinux enforcement. Thus, the target device has to either run a user-debug build of Android or be rooted (e.g., for devices of other OEMs).

Collecting Public APIs ($\boldsymbol{RQ}_1$): DYNAMO implements a generic technique for discovering public APIs of target devices. Specifically, the `TS` uses Java reflection to retrieve all Binder handles from Android's `ServiceManager`, casts them to their corresponding proxies where all methods are defined, and collects all methods' signatures. The `TM` pulls and saves this list for each unique target device. Another relevant information that is collected at this point is the mapping between services and hosting processes, identified by Linux PIDs. This is important information for instrumenting the APIs, constructing the feedback channel, and impersonating different calling contexts.

Selecting API for Testing: We designed DYNAMO such that it allows several devices running the same Android version to work collaboratively on the same API list to speed up the analysis. Each device locks and starts processing one API at a time. Each tested API has a configuration file that describes the API (e.g., method's signature and hosting process) and the testing configuration (e.g., initial seeds, strategies, log of testing iterations). This file is responsible for the deterministic behaviour of the tests and the ability to reproduce results.

Generating Input ($\boldsymbol{RQ}_2$): DYNAMO extracts the input types from the API's signature and assigns each primitive Java parameter (e.g., string, int, boolean) a short list of predefined seed values. The list of seeds is manually created before the testing begins (see Appendix E for the complete list). For instance, a parameter of string type is assigned a seed list that contains null, empty string, the package name of `TS`, other package names, and a random permission name. Since not all seed values might be semantically relevant to the tested API, this module further leverages static analysis of the source code (e.g., from AOSP) to refine the seeds by heuristically selecting relevant seed values based on the parameters' names of the API's signature and excluding others from the predefined seeds list. For instance, an API that uses a string parameter that is named *packageName* or *pkg* would take a few valid package names as initial seeds for that parameter instead of testing non-relevant values. Experiments show that this technique for refining inputs reduces the testing time by at least a factor of 5 without affecting the modeled policy.

However, there are other non-primitive input types that we classify in two categories: basic Android types (e.g., Intent and URI) and complex types (e.g., event listeners and bitmap). Similar to primitive types, we define fixed seeds for basic types that are used to instantiate objects of those types at runtime. For instance, a package name and a class name are used as seeds for creating an Intent object. For complex types, the `TS` uses a recursive algorithm that receives the qualified class name of the object to be instantiated as an input, searches its class for a constructor or a method that receives inputs of primitive types and returns an object of the desired complex type, and calls that constructor/method to instantiate the object. However, if the constructor or the method itself receives objects of complex types, the algorithm tries to instantiate those objects first, and so on. If the object

Algorithm 1 Building Security Policy of an API

```
Input Service s, API a, ReachableMethods m
Output modeled security policy
 1: procedure BUILDSECURITYPOLICY
 2: strategies := getDefinedStratigies()
 3: inputs := getInputSeeds(s, a)
 4: results := []
 5: for strategy of strategies do
 6:     for input of inputs do
 7:         privileges := []
 8:         repeat
 9:             instrumentReachableMethods(m)
10:             instrumentSecurityCheckingMethods()
11:             missingPrivileges = invokeAPI(s, a)
12:             privileges.insert(missingPrivileges)
13:             r := collectTracesAndResult(s, a, privileges)
14:             results.insert(r)
15:             measureAndReportCoverage()
16:         until missingPrivileges is empty
17: modelSecurityPolicy(s, a, results)
```

is an implementation of a specific interface, the `TS` uses Java reflection to build a proxy object that implements that interface. This input generation technique has proven to be effective, except for a few special classes that can be instantiated but not used in IPC due to early sanitization while marshalling inputs in the proxy. The lists of seeds for API parameters are then used to create the Cartesian products of seed inputs, where each product is called an *input set*. Those input sets are stored in the configuration file of the API. When the `TM` selects an input set to be tested, this set gets encoded in a file that is pushed to the *TS* (❶), which uses it to generate API's input.

Measuring Coverage ($\boldsymbol{RQ}_3$): As we will discuss later in this section, one important feature that DYNAMO relies on for modeling the security policies of APIs and measuring coverage are the methods' execution traces that are collected when invoking an API under test. To achieve this, DYNAMO conducts reachability analysis with WALA [3] on the target API and produces a list of qualified method names that are reachable from the target API. DYNAMO uses this list to instrument the methods such that they report their stack traces to the `TM` when called. This also serves as feedback on achieved coverage for each testing iteration. Specifically, we measure the coverage by counting the number of unique traces and comparing it with the total number of unique methods from the reachability analysis. As such, DYNAMO uses method-coverage as a proxy for code coverage. A low method-coverage of an API could suggest a failing execution due to input sanity checks, an unexpected new class of security checks, or an early return. As such, the low method-coverage indicates the need to refine seeds, elevate input generation techniques, or investigate new security checks.

Clearly, another approach to using WALA is instrumenting all methods of the framework. Unfortunately, our experiments show that this technique comes with a huge overhead that destabilizes and hinders the analysis and therefore was not technically possible in the current version of DYNAMO. Unfortunately, relying on Chizpurfle's approach for measuring code-coverage [34] was also not possible due to technical limitations in the underlying technique (i.e., Frida's Stalker [46]), which does not work on all CPU architectures of tested devices. Chizpurfle's approach also entailed high overhead that hindered our analysis as the execution of several threads needed to be tracked (i.e., from processes of `system_server`,

```
1 public long addClient(IAccManClient cb, int userId) {
2     final int resolvedUserId =
       resolveCallingUserId(userId);
3     {...}
4 }
5 public int resolveCallingUserId(int userId) {
6     int resolvedUserId = resolve(userId);
7     int callingUid = Binder.getCallingUid();
8
9     if ( callingUid == 0 || callingUid == 1000 )
10        return resolvedUserId;
11
12    int callingUserId = UserHandle.getUserId(callingUid);
13    if ( callingUserId == userId )
14        return resolvedUserId;
15
16    if( ! checkPermission(INTERACT_ACROSS_USERS,
       callingUid) )
17        throw new SecurityException();
18
19    return resolvedUserId;
20 }
```

Listing 1. Simplified illustration of the `addClient` API

```
CallingUid == 0 || CallingUid == 1000 ||
 CallingUid.userId == Parameter#2 ||
 CallingUid.hasPermission(INTERACT_ACROSS_USERS)
```

Listing 2. Modeled security policy of the `addClient` API

`TS`, and `Target Service`). Relying on WALA's results for reachability analysis for measuring method-coverage, given that they might be imprecise, does not affect the precision of our approach as we filter out stack traces that do not start with the target API (i.e., are not triggered by invoking the API).

DYNAMO*'s Testing Strategies (**RQ**$_4$):* To model the access control of a specific API, DYNAMO employs several predefined testing strategies (see Algorithm 1). A *strategy* is a set of operations that try to discover one aspect of the security policy of the target API. For instance, one strategy could focus on discovering permissions while another detects checks on caller's UID and PID. Each strategy starts with a list of input sets that are generated from predefined seeds. The same list is exercised in all strategies. For each strategy, DYNAMO exercises each input set and detects the failing security checks as they occur. When security checks are reported, they are fed back in the next iteration for the same input set where DYNAMO instructs the `IS` to bypass failing checks to detect other checks along the same execution path. This process repeats until no further security checks are reported. After all strategies are concluded, the `TM` marks the current API as complete and moves on to the next API to be tested. With this feedback-driven testing, DYNAMO explores several caller contexts (i.e., third-party app, privilege app, etc.) to trigger and detect security checks.

Example: To understand how DYNAMO's strategies work in detail, consider a simplified version of the `addClient` API from the `AccessibilityManagerService` as exampled (see Listing 1). To execute this API, the caller has to qualify for one of the following conditions: 1) The caller must run under a UID equal to `0` or `1000` (line 9). 2) The caller must run in a context whose `userId` is equal to the API's second input parameter (line 13). 3) The caller must be granted the `INTERACT_ACROSS_USERS` (`ACU`) permission (line 16). In case none of those conditions is satisfied, a `SecurityException` is raised (line 25). To simplify the example, we define only one set of inputs that consists of `null` and `10` as first and second parameters, respectively. Choosing `10` for this example is not arbitrary as it corresponds to the `userId` of the secondary profile (while `0` is the `userId` of the main profile). Invoking the API with this set of inputs from the main profile and from an unprivileged context would result in the caller receiving a `SecurityException` because none of the above conditions would qualify. In the following, we explain how DYNAMO detects those conditions.

Since missing permissions is the most common cause for denying access, DYNAMO's first strategy aims for discovering and granting the missing permission(s) to the calling context. DYNAMO leverages the fact that permissions are centrally checked by the `checkUidPermission` API of the `PackageManagerService` and therefore instruments this API to report the missing permission(s) to the `TM` (see steps ❷ and ❸ in Figure 2). This API receives the permission name and caller's identity (i.e., UID and PID) and returns either `GRANTED` or `MISSING` based on whether the caller (identified by its UID and PID) has been previously granted this permission or not. After reporting the missing permission to the `TM` through the `Analyzer Module` (❸, ❻ and ❼), the permission is used in the next testing iteration to elevate the privileges of the caller's context by again instrumenting the `checkUidPermission` API such that it would return a `GRANTED` response when called for the same permission and caller's identity (❷ and ❸). This way, DYNAMO manages to report that `ACU` permission is needed for unprivileged callers of the main profile. This strategy generally detects all method-based checks that are known from the literature (AppOps permissions, checks on calling package, or user restriction checks). Notice that this strategy is orthogonal to all other strategies, for example, to detect permissions from secondary profiles (see below).

The second strategy tries to infer cross-user permissions. Thus, the `TM` instruments the system such that the caller would impersonate a UID from the secondary profile (which starts at `1000000` [23]). To technically achieve this, the `TM` instructs the `IS` to instrument the `Binder.getCallingUid()` of the Binder class on the target service such that it would return the impersonated instead of the actual UID of the caller (❷ and ❸). When calling the API with the same input set used before while impersonating the new context, the `callingUserId` will be resolved to `10` (which corresponds to the `userId` of the second profile) and that matches the second parameter (see lines 13). Therefore, the API will successfully execute without requiring any permission. Notice that faking the PID is technically similar to faking the UID (i.e., via instrumenting the `Binder.getCallingPid()`). Lastly, DYNAMO tries to detect security checks based on the caller's UIDs (see line 9). For that, it harvests all UIDs that are reserved for system users via the shell interface of the package manager (i.e., based on the condition `0 <= UID <= 2000` [23]). Thus, when invoking the API while the caller is impersonating root (`UID = 0`) or system (`UID = 1000`), the API will successfully execute. Otherwise, `callingUserId` would resolve to `0` and a `SecurityException` will be raised if the privileged UID does not have the specified permission.

Throughout the testing phase, the configuration file of the API under test serves as a reference to identify the current strategy and state (e.g., permissions granted or the different impersonated contexts). For each strategy, the API is executed until no new changes on the calling context are required (e.g., no need to grant a new permission). When that is satisfied, the API is executed one more time, and the execution of the strategy is considered successful if the results of the last two execution iterations match entirely. When all strategies are exercised, the `Analyzer Module` concludes the testing phase for the current API (❽).

*Instrumenting Targets (**RQ_5**):* In addition to instrumenting the methods that implement each strategy (as explained above), the `TM` further instruments the reachable methods from the target API (i.e., based on WALA's reachability analysis) such that they report their stack traces, inputs, and outputs to the *Analyzer Module*. A stack trace is considered relevant only when it starts with the target API and when the caller's PID matches `TS`'s PID (i.e., triggered by DYNAMO's testing).

Executing API: To execute one testing iteration, the `TM` sends a message to the `TS` so it would generate API's inputs (as described earlier) and invoke the API using Java reflection (❹). The API will then trigger the instrumented security checks and reachable methods (❺) and they will start reporting to the *Analyzer Module* (❸, ❻, and ❼). After a specified timeout, the `TM` reverts instrumented methods to their original implementation and pulls the invocation results from the `TS` and feeds it to the *Analyzer Module*.

*Modeling of Permission Mapping (**RQ_6**):* While some security checks are easily inferred (e.g., permissions and AppOps), others require multiple invocations of the API under different contexts to be inferred. Take the cross-user security checks from Listing 1 as an example (line 13). We can infer the presence of a multi-user check only after looking at the results of invoking the API from the secondary profile and the results from the main profile while no permissions are assigned to the caller. The call from the main profile failed with a security exception, while the call from the second profile succeeded. This suggests the presence of a cross-user check but is not definite in describing this check because it could be based on input or whitelisting the calls from secondary profiles (e.g., were `callingUserId` checked to equal `0` instead of `userId`). To differentiate, we observe the API's result when its second parameter changes. When the second parameter is `0`, the outcome of invoking the API from the main and secondary profile is flipped (i.e., allowing a call from the main profile and requiring permission from the secondary profile). However, when it is neither `0` nor `10`, both calls start asking for the `ACU` permission. At this point, we know that the API's second parameter has an influence on access control and report it accordingly. We apply a similar mechanism for detecting whitelisted UIDs. Specifically, we observe the API's behaviour and traces when invoking the API while impersonating the root and system UIDs to find that `resolveCallingUserId` returns without raising an exception, suggesting the presence of disjunct checks on those UIDs. Based on that, the `Policy Builder` summarizes the results in a first-order formula that models `addClient`'s security policy as shown in Listing 2.

Although not covered in the example, we also detect further security checks along the execution paths and report those checks in conjunction with others. We handle the special case where any of the multiple permissions is needed. For that, we leverage that all permissions would be checked before throwing a `SecurityException` when the last permission check fails. This means multiple permissions are reported. Thus, the `TM` exercises those permissions in disjunction to observe the relation between them. When the traces include security checks, but our rules fail to model the policy, we manually look at the traces and try to enhance our rules. Despite that, with a few predefined association rules between API calls and traces, we can model the security policy of the majority of APIs (see Section V). Of course, we realize that this modeling can be incomplete, as we discuss in Section VI.

V. Comparison with Prior Results

We built DYNAMO to complement the methodology for studying the security policy of Android's application framework and evaluate the results of prior works of this domain. Thus, we evaluate our tool in four prominent use cases from the literature. We start by comparing the state-of-the-art permission mapping from Arcade [5] with the mapping built by DYNAMO (Section V-A). Second, we reproduce and evaluate the permission re-delegation vulnerabilities, as reported by ARF [29] (Section V-B). The first two evaluations show that DYNAMO can confirm the majority of prior results but can also be used to extend and refute (or at least question) several others. Third, inspired by Kratos [47], we conduct a lightweight analysis for detecting access control inconsistencies between APIs of system services (Section V-C). We were able to discover several unprotected and misconfigured APIs. Fourth, we use DYNAMO's permission mapping for the latest Android release (v10) to assess the completeness and correctness of APIs' permission information in the developer's documentation of Android (Section V-D). We found several cases where the official documentation is incomplete or inaccurate, affecting third-party app developers.

Evaluation environment: We tested DYNAMO on different Android versions (6, 8.1, 10, and 11) of vanilla Android and other vendor images. We used a combination of hardware devices (Pixel 4, Nexus 5, and One+) and emulators (Cuttlefish [1] and Android Studio emulators) running different CPU architectures (x86, x86_64, arm, and arm64). DYNAMO, was fully functional under all of those settings.

A. Evaluating Previous Permission Mappings

Static analysis has been shown to be capable of building good approximations of permission mappings [11], [12], [5]. However, since there is no systematic approach for verifying those mappings at scale, we cannot easily describe their accuracy and completeness. Using DYNAMO, we are finally able to systematically evaluate previous mappings using dynamic testing. The most up-to-date mappings are from Axplorer and Arcade. Since all but 20 protected APIs reported by Axplorer are also covered in Arcade, we only focus on the state-of-the-art mapping from Arcade.

Overview. As shown in Figure 3, DYNAMO is able to verify the permission mapping for 76.1% of 951 APIs also reported by Arcade and to extend the mapping by 10.6% with newly discovered security checks which we consider as

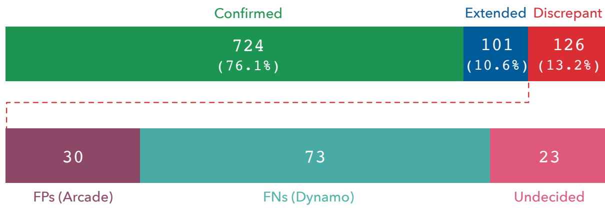

Fig. 3. Breakdown of results from evaluating 951 common APIs from Arcade.

```
1 public List<?> getScanResults() {
2     int callingUid = Binder.getCallingUid();
3     long ident = Binder.clearCallingIdentity();
4     enforcePermission(ACCESS_FINE_LOCATION, callingUid);
5     Binder.restoreCallingIdentity(ident);
6 }
```

Listing 3. Enforcing permission after clearing the caller's identity

False Negatives (FNs) for Arcade. Manual code analysis of the 13.2% discrepant APIs identified 30 FPs in Arcade's mapping and 73 FNs in DYNAMO. DYNAMO was also able to report security checks for 343 APIs that are missing in Arcade's mapping (FNs in Arcade) but also missed 247 APIs reported in Arcade (FNs in DYNAMO). We shared our findings with the authors of Arcade, and we incorporate their feedback in our root-cause analysis for the discovered discrepancies. As mentioned earlier, we shared our findings with the authors of Arcade who reacted to our discussion and updated their tool to increase API-coverage and fix some erroneous reports to their old permission mapping. As such, the updated permission mapping from Arcade reports new access control information for 217 APIs. We discuss those changes separately at the end of this section.

Setup. To avoid wrong reporting due version mismatch, we fixed DYNAMO to analyze Android-6.0.1_r10 (v.23), the same build used for Arcade's mapping. We used two Nexus 5 devices and four x86 emulators to run the analysis (using Cuttlefish was infeasible as it requires Android API level $\geq$28). The analysis took about four weeks to conclude for this test setup.

Scope of Analysis. Table II describes the permission mappings in this evaluation. DYNAMO analyzed 2,057 public APIs and reported security checks for 1,294 (62.9%) of them while Arcade analyzed 4,189 APIs and reported similar security checks for 1,198 (28.5%) of them. DYNAMO and Arcade reported 2,064 and 2,164 execution paths, accordingly. As expected, permissions are the most common security checks with 160 unique permissions enforced in 84.3% of the protected APIs found by DYNAMO. While missing in all previous permission mappings, DYNAMO found AppOps checks (e.g., package validation) in 16.3% of the protected APIs it covers. Both permissions mappings that are used in this evaluation are included in the file `android-6.0.1_r10.json` under the `results` folder in DYNAMO's project repository [16].

There are 951 APIs that are shared between both mappings. This means that DYNAMO and Arcade exclusively reported 343 and 247 unique APIs, respectively. We found three reasons why DYNAMO missed mappings for 247 APIs that exist in Arcade's mapping: First, DYNAMO considers APIs whose services are deployed on the target device and can be *directly* retrieved from the `ServiceManager`. However, some services are wrapped inside other services and require a multi-step approach to retrieve them [38]. We only discovered this after the testing was concluded and, therefore, DYNAMO missed 131 APIs. Additionally, the low code coverage and missing calling dependencies led to missing another 113 APIs. Finally, DYNAMO did not report any security checks for 3 other APIs in comparison to Arcade. Manual code inspection revealed that those APIs are actually unprotected and Arcade's reporting is inaccurate (i.e., FPs). Since Arcade's source code was not available to us, we could not investigate the causes of this wrong reporting. Similarly, we investigated the reasons why Arcade missed 343 APIs reported by DYNAMO. First, Arcade did not report mappings for some services, such as *WiFi* and *Telecom*, which account for 65.3% of the missed APIs. Arcade's authors acknowledged this shortcoming and attributed it to not including the corresponding binary files for all services in the analysis. However, the remaining missed APIs from Arcade belong to services whose corresponding binary files were considered. This suggests other technical limitations of Arcade, such as early termination of the analysis, inability to identify entry points to APIs, and improper bridging of IPC, which we discuss later in the results in this section.

Comparison. For the 951 common APIs, we defined strict criteria that classify an API into one of three categories based on how DYNAMO's mapping compares with Arcade's mapping for that API (see also Figure 3). Those categories are 1) exactly matching if both mappings have the same number of paths and enforce the same security checks on each path, 2) APIs with new security checks if DYNAMO reports more paths or security checks than the ones reported by Arcade, and 3) discrepant security checks if DYNAMO's mapping has different or less checks. This third category is further classified into three subcategories: a) FPs from Arcade, b) FNs from DYNAMO, and c) APIs that we could not classify as any of the other subcategories because the APIs' logic was too complex to be manually analyzed to search for checks reported by Arcade.

Results. As shown in Figure 3, this automatic comparison reported 724 APIs (76.1%) with exactly matching security checks. We consider those APIs as confirmed mappings given that dynamic results are sound. Zooming onto this number reveals that 89.6% of the confirmed mappings contain one or two paths with one path being dominant. The confirmed APIs also enforce a combination of 117 unique permissions. The comparison also showed that DYNAMO was able to extract more security checks than Arcade for 101 APIs, which was surprising given that static analysis is known for achieving high coverage. In total, 51 new paths were discovered by DYNAMO as well as 23 new unique permissions. After manual code inspection and a discussion with Arcade's authors, we can attribute this particular discrepancy to three issues: 1) Arcade relies on a simplified and incomplete mechanism for bridging IPC [11], [47] and subsequently misses some permissions enforced in APIs invoked via IPC from within the analyzed API. 2) Arcade relies on heuristics to identify the endpoints of the analysis and avoid noise in the results to retain its approach's practicality. For example, they stop the analysis immediately before the caller's identity is cleared with `Binder.clearCallingIdentity`. However, this leads

TABLE I. SUMMARY OF PERMISSION MAPPINGS BUILT BY ARCADE AND DYNAMO.

Tool	Android Version	# Covered APIs	# Protected APIs	# Paths	# Permissions	# APIs Enforcing *			
						Permissions	UID/PID	AppOps	Others **
DYNAMO	Android-6.0.1_r10	2,057	1,294 (62.9%)	2,164	160	1,092 (84.3%)	302 (23.3%)	212 (16.3%)	108 (8.3%)
Arcade	Android-6.0.1_r10	4,189	1,198*** (28.5%)	2,064	170	1074 (89.6%)	281 (14.9%)	1 (0%)	34 (2.8%)
DYNAMO	Android-10.0.0_r27	3,579	2,537 (70.8%)	3,877	257	1,953 (76.9%)	392 (10.9%)	489 (19.2%)	370 (14.5%)

* Different checks can be used for the same API; ** Checks if caller is isolated or in same app, user restrictions, and unhandled security exceptions.
*** Arcade reports 1,519 protected APIs in the paper, however, the published mapping used in this evaluation includes only information for 1,198 APIs.

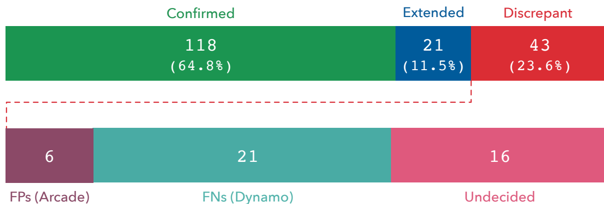

Fig. 4. Breakdown of results from evaluating 182 common APIs between DYNAMO and the added APIs in Arcade's updated permission mapping.

to missing security checks performed *after* the identity is cleared. For example, in Listing 3, the caller's UID is saved in line 2 before it is cleared in line 3. The original UID stored in variable `callingUid` is then used to check for the permission in line 4. Consequently, this permission is not detected by Arcade. 3) Arcade reduces the Control Flow Graph (CFG) of an API to an abstract CFG that only contains security checks. This design decision of Arcade is error-prone as Arcade might not be able to infer the security relevance of a check on API's access control—either due to limiting the analysis or inability to resolve runtime values—and therefore truncates it.

For the remaining 126 APIs, the two mappings report different security checks and require manual inspection to uncover the reason for this discrepancy. Of those, we confirmed with clear evidence 30 APIs as FPs by Arcade (Appendix A), confirmed 73 APIs as FNs by DYNAMO (see Appendix B for examples), and for 22 APIs we could not classify them as either of those previous cases despite the high manual effort. Among the confirmed FPs from Arcade, 13 APIs were mapped with permissions that are actually not enforced by the APIs and 17 APIs actually enforced different permissions than reported by Arcade. We shared a representative sample of FP cases with Arcade's authors who confirmed them. As for the APIs that report different permissions, Arcade's authors attribute those cases to mistakes in string resolution of the permission name. Since we do not have access to Arcade's tool, we cannot further investigate the reasons behind reporting non-existing permissions. But, we speculate that the permission mapping of some of those APIs has been mistakenly mixed with other APIs that enforce the same permissions. Out of the 30 FPs from Arcade, we found that 12 APIs are protected by permissions that can be requested by third-party app developers. For those APIs, Arcade's mapping suggests the need for more permissions than actually needed. Clearly, this aggravates the problem of over-privileged apps instead of mitigating it and entails negative consequences that we discuss in Section VI.

Updated Mapping From Aracde. As a result of sharing our findings with Arcade's authors, the authors released fixes to the Arcade tool and updated their published permission mapping. The update focused on increasing the API-coverage and fixing erroneous mappings. As such, the updated permission mapping from Arcade lists new access control information for 217 APIs. Of those APIs, we found 182 APIs that also exist in the mapping from DYNAMO. The remaining 35 APIs are missing from DYNAMO for the same reasons discussed earlier (e.g., low code coverage, failing to invoke the API due to unsatisfied conditions, etc). We applied the same methodology discussed earlier for evaluating the 182 common APIs between both mappings (i.e., automatic comparison and then manual code inspection). A summary of this evaluation is presented in Figure 4. As shown in the figure, 118 of the 182 common APIs are completely matched between both permission mappings (i.e., confirmed), 21 APIs are extended by DYNAMO (i.e., FNs in Arcade), while 43 APIs exhibit discrepancies between the reported permission mappings. After manually inspecting the discrepant APIs, we found 6 FPs in Arcade that report wrong association between permissions (e.g., required permissions are reported in conjunction to each other instead of disjunction). We also found 21 FNs in DYNAMO and 16 APIs that we could not fully analyze due to time constraints. It is worth noting that we could not identify new root-causes for the discrepant APIs in addition to those identified earlier. Overall, the updated permission mapping from Arcade added 171 APIs with manually verified permission mappings (i.e., 118 confirmed, 21 FNs in DYNAMO, and 32 missing APIs from DYNAMO)

B. Verifying Permission Re-Delegation Vulnerabilities

System APIs in Android are highly interconnected and rely on each other. Since there is no specification for the security policy of those APIs, discrepancies in their access control can arise. For example, one API (deputy) might expose the functionality of another protected API (target), creating a less-restrictive path to execute that functionality as opposed to the path that directly leads to the target API. This problem is known as permission re-delegation. ARF [29] analyzed Android's framework for permission re-delegation vulnerabilities. It used static analysis and manual code inspection to discover vulnerable paths in Android-8.1.0_r1. The authors reported 88 APIs (out of 170 discovered) that are potentially vulnerable. The authors were transparent in their reporting, stressing that they have not created proofs of concept attacks based on those vulnerabilities. We saw this as an opportunity to showcase DYNAMO's ability in verifying the vulnerabilities using dynamic testing.

We used DYNAMO to analyze 33 of the 88 APIs reported by ARF. We chose those APIs because, according to their vulnerability description, they have the highest impact on user privacy and experience. We compared the observed runtime behaviour with the vulnerability description, and we were able to discover discrepancies in 5 APIs. Based on correspondence with ARF's authors and on manual code inspection of those APIs, we uncovered two reasons that lead to mistakenly reporting those APIs as vulnerable. First, ARF was not able to correctly resolve the identity used in permission checks. Second, even manual analysis is limited and cannot cover all aspects of access control. While the first reason is indeed an inherent limitation of static analysis, authors' assumptions that were incorporated in the vulnerability analysis (i.e., how caller's identity is resolved and propagated) aggravated the problem and lead to this imprecise reporting. For example, one API (`whitelistAppTemporarily`) that acts as a deputy is mistakenly assumed to escalate caller's privileges to those of the system by overwriting the caller's UID with the system's UID before invoking the target protected API (`addPowerSaveTempWhitelistApp`). However, dynamic testing showed that both APIs enforce the same permission and the caller's identity was not overwritten.

The more complicated task was confirming the remaining vulnerabilities. Our strategy was as follows: we translated the described vulnerability into different sets of parameters, which, when tested, would trigger the vulnerability. For example, if the API exposes information across different profiles without protection, we craft inputs and fake the calling context to mimic a call from the same and a different profile. Using DYNAMO, those different test settings were easily configured and set up. We compared the results (i.e., returned values and traces) of different testing iterations and considered the vulnerability confirmed if both calls from the same and different profile would yield the same output. For APIs that do not return values, we made a decision based on the collected traces and GUI feedback. This simplified verification strategy enabled us to confirm 10 vulnerabilities with minimal manual effort. According to the authors, four of those vulnerabilities received CVEs from Google. Unfortunately, we could not make a decision on the remaining 16 APIs because the collected traces and GUI feedback of those APIs were inadequate.

C. Inconsistency Analysis of APIs' Access Control

Inspired by Kratos's [47] approach, we used the permission mappings, APIs' traces, and contextual information produced by DYNAMO for Android 6 and 10 to conduct a lightweight inconsistency analysis. Our strategy is simple but effective as it discovered 5 sensitive APIs that are unprotected and 65 APIs with permission misconfigurations. We used the traces collected for all APIs of the same Android version to detect execution paths from different APIs that lead to the same sink but enforce different security checks. Since identifying the exact sink is challenging and a problem in its own [12], we heuristically created a list of sinks for each API. We then search for different APIs that share sinks and compare their permission information as reported by DYNAMO, including the identity for which the permission is enforced. Unfortunately, this simplified mechanism creates a lot of noise, which we circumvent by creating a threshold of matching sinks to decide if two APIs are sharing the same sinks. When the comparison of permissions of two APIs produces a mismatch, we flag the APIs for manual code inspection to see if both mappings are different but practically enforcing the same level of access control (i.e., no vulnerability) or if one is less restrictive than the other (i.e., possible vulnerability).

As a result, we found one case in the latest Android release (Android 10) between the `getBoolean` (`GB`) and `isTrustUsuallyManaged` (`ITM`) APIs of `LockSettingsService` and `TrustManagerService`, respectively. The `ITM` API queries an internal storage using a key to check if a specific profile has enabled and configured trust agents [25]. This API is protected by system-level permission. On the other hand, the `GB` API accepts an arbitrary string and uses it as a key to retrieve the corresponding value from the same storage used by `ITM`. When the `GB` API is used with the same key from the `ITM` API, the call to `GB` successfully retrieves the protected value without enforcing any security checks. This inconsistency was confirmed and fixed by Google. We detected four more cases in Android 6; however, those cases were already fixed in later Android releases and apparently are known to Google. Those APIs are `getDeviceIdForPhone`, `getLine1NumberForSubscriber`, `getLine1Number`, and `getDeviceId` of the `PhoneInterfaceManager`. Additionally, we discovered 65 APIs from the `ActivityService` in Android 6 where the permissions are enforced against the identity of the app running on the main profile while the call can originate from a secondary profile. In other words, given that an app can exist in main and a secondary profile where each instance can have a different set of permissions, the app from the second profile—which maybe unprivileged—can still call those affected APIs if the app on the main profile is granted those permissions. Those cases illustrate the intricacies when multiple policies need to be stacked (here, permissions and profiles).

D. Evaluating Android's Developer Documentation

Android's developer documentation has been shown by prior works [20], [51] to be imprecise and incomplete. Since the documentation has evolved since these prior works, we reassess the current documentation for the latest Android release Android-10.0.0_r27 (API level 29). Our evaluation shows that the permission mapping from the documentation is incomplete for 66 APIs and contains errors for 9 APIs out of 439 automatically tested APIs. On the other hand, we found that the permission information extracted with DYNAMO matches the documentation for 40.5% of those tested APIs.

To build our permission mapping for the targeted Android version, we used special virtual devices from Google, called Cuttlefish [1], which allowed us to scale up the analysis by creating 20 VM instances on Google's Compute Engine with one Cuttlefish emulator per VM. Each instance is equipped with 4 vCPUs and 15GB of RAM. The analysis took two weeks and we were eventually able to analyze 3,579 different APIs from 121 system services and produce permission mapping information for 2,537 APIs. Table II includes the mapping results for this Android version (row *Android-10.0.0_r27*).

To collect the permissions documented by Google, we leverage the fact that the online documentation is automatically derived from the source code and, therefore, extract the

mapping from the source code directly. We search for APIs that are well annotated with required permissions through the `@RequiresPermission` (`@RP`) tag. Those APIs can then be automatically collected using static code analysis. This has the benefit of analyzing a structured target using very simple static analysis techniques as opposed to text-analysis when working directly on the developer documentation. However, since the developer's documentation only lists the APIs of the manager classes (e.g., `WifiManager`) which then call to remote services (e.g., `WifiServiceImpl`) but DYNAMO builds the mapping for the *service* APIs, we need to map the APIs from the manager classes to the APIs from the services. We used Soot [50] to extract the methods with the Java annotation `@RP` from manager classes. For each extracted method, we first collect the required permissions and their relation. Then, we connect the `IBinder` calls from within the manager class to matching remote APIs of the services, which were extracted at runtime from all services. We were able to collect 301 APIs and their annotated permissions and map them to their remote service APIs. However, evaluating the documentation also requires detecting cases where the documentation is incomplete or missing a permission. In contrast to collecting annotations, this would require sophisticated text processing to automatically determine the *absence* of permission documentation. Thus, for this evaluation, we limit our assessment of missing documentation to three randomly picked big services as case studies, for which we manually analyze the documentation to detect the absence of documented permissions. Those services are `WifiService`, `ConnectivityService`, and `AudioService`. We manually analyzed the corresponding manager classes and collected 138 methods that call remote services but were *not* annotated with `@RP` nor have a description of required permissions.

Out of the 301 APIs that were automatically collected, we found 178 APIs (59.1%) whose permission mapping by DYNAMO matches the mapping from the source code annotation. Those APIs collectively enforce 64 unique permissions. It is worth noting that DYNAMO managed to extract more permissions than those annotated in the source code for 9 of those APIs. This is because the managers' APIs are annotated with a subset of permissions that are required for one chosen execution path depending on the input (i.e, parameter-sensitive enforcement), while DYNAMO extracted permission mappings for other execution paths as it bypasses the managers' APIs and directly tests the corresponding service APIs. However, we found that DYNAMO failed to construct the permission mapping for 50 APIs due to either low code coverage or failing to invoke the API because of a missing dependency. The remaining 44 APIs revealed three error patterns in the annotated permissions in the source code: First, the source code defines required permissions that are not actually enforced (9 APIs). Second, the source code defines the permissions required for only some execution paths, while the user-controlled input can trigger all execution paths (29 APIs). Third, the source code only partially describes the API's required permissions in the `@RP`, causing the matching process to fail. However, after manually checking the source code, we find complementary information about the needed permissions (6 APIs). Surprisingly, after analyzing the 138 APIs from the three manager classes that do not have any sort of information regarding the required permissions, we found 31 APIs with missing information that, in fact, require a permission from the caller. We reported all of our findings to Google and they are in the process of taking corrective actions for their developer documentation. Appendix D details our results.

The root cause analysis on why those inconsistencies and incomplete annotations exist in the documentation are three-fold. First, the documentation is best human effort. Second, while the services evolve to more or new required permissions, the manager APIs fall behind and are overlooked, especially for the APIs that are hidden from third-party app developers. Third, the annotations are too inflexible to convey the parameter sensitivity. As a result, they are used to reflect the permissions required for one execution path only and further conditions are, sometimes, mentioned in natural language only.

VI. DISCUSSION

We discuss our evaluation and try to provide lessons learned for analyzing Android's application framework.

Security Impact. While DYNAMO's improvements to the documentation (Section V-D) and the discovery of unprotected and misconfigured interfaces (Section V-C) were acknowledged by Google as direct results of this work, we see another long-term security impact from the permission mapping that we generate with DYNAMO. Accurate and complete permission mappings are essential in various security applications in Android, such as detecting over-privileged apps [44], [40], detecting malware [9], [18], [27], [36], [48], [57], finding inconsistencies within the application framework [4], [6], [28], [29], [47], [56], or others [13], [35], [43], [51]. For instance, developers without complete and accurate mappings tend to request more permissions than necessary for their apps. This is not only a nuisance for developers, but has been shown to frustrate the user [39], [49] and cause a feeling of erosion of privacy [37], [31]. Google recently even started nudging developers of over-privileged apps into rethinking their apps' behavior [45], [44]. Moreover, malware detection depends on accurate permission mappings to identify dangerous calls in apps as part of the feature sets of their classifiers [9], [27], [36], [57]. Without complete mappings, malicious apps might be missed (e.g., by not detecting usage of sensitive protected APIs) while imprecise mappings can lead to erroneously flagging benign apps as malicious (e.g., confusion about what would be the right set of permissions for a benign app). Lastly, considering DYNAMO's approach to dynamically test the application framework API, we envision extending DYNAMO into a tool for systematically testing the application framework's internals for inconsistencies (Section V-C) and for verification (Sections V-A and V-B) in the future.

Soundness and Completeness of DYNAMO's Results. The results of dynamic analysis are sound for the executed paths but can be incomplete for some parameter-sensitive APIs. In our results, we find that some security checks were overlooked because we only rely on our observations (e.g., outputs, traces, and reported checks) that we encode as association rules to automatically decide on the security exception. For example, some APIs would throw a security exception that requires a human interpretation to identify the check that causes the exception and, therefore, cannot be automatically detected. A more reliable approach would be looking at those traces

manually, which is not scalable, or to devise better automatic classification of traces (e.g., using learning techniques).

Benefits of Combined Approaches. One thing that our results show is that neither static nor dynamic analysis alone can uncover the intricate details of the framework and that—in alignment with other software testing domains—a combined approach is needed. Take the problem of building permission mapping as an example. The common assumption so far is that the compromises made by static analysis—e.g., limiting the analysis to cater for performance—can still yield better results than dynamic testing that suffers from poor code coverage. However, we have demonstrated that even simple techniques for achieving high coverage in dynamic testing can outperform static analysis for some cases. When considering our comparison with Arcade, DYNAMO was able to complement the permission mapping for 444 APIs that are missing or incompletely modeled by Arcade in comparison to 320 APIs that are missing or incompletely modeled by DYNAMO (becomes 283 and 373 APIs, respectively, when comparing with the updated permission mapping from Arcade).

The limitations and strengths of both approaches that complement each other serve as an indication that moving for a hybrid approach is the next natural step in improving analysis, and we find concrete evidence in our results: (1) Arcade limits the scope of analysis to preserve their practicality by only considering checks before the identity is cleared (see Listing 3), thus missing subsequent security checks. DYNAMO reports all security checks for the selected execution path and helped to find this shortcoming, which now can be improved for future static analyses. (2) Dynamic analysis fails in detecting complex access control logic and inferring the relation between the input and corresponding execution paths. This, on the other hand, is possible via static analysis with minimal overhead. (3) By design, existing static analysis works cannot analyze native APIs because they rely on Java-only static analysis framework (i.e., [3], [50]). In contrast, dynamic testing uses IPC primitives that do not differentiate between the implementation language of the target API. (4) Static analysis thrives on assumptions based on observations that are mainly collected from manual code analysis. This can lead to wrong results, as we have demonstrated for some of ARF's results. Feeding back our observations, future static analysis can be improved with this new expert knowledge.

Verifiability of Static Analysis Results. Being able to verify the permission mappings from static analysis solutions is crucial to the development of their approaches, for example, by having direct feedback that confirms the results and avoids huge manual efforts for result verification. As an example, PScout [11] followed this general advice, but their verification was based on only a heuristic evaluation that was generally imprecise. A tool like DYNAMO that leverages advances in dynamic testing and runtime instrumentation can fill this gap. This is exemplified by 76.1% automatically confirmed cases by DYNAMO and the extended and new APIs that have been previously overlooked and whose root cause analysis yielded valuable insights for static analysis.

Further, being able to identify the location of the security checks, the order of checks, the paths that lead to them, and the resolved variables used in them is crucial to properly analyze and understand the framework. This information is easily collected by DYNAMO for the executed paths but poses a huge challenge for static analysis. For instance, one of the wrong assumptions that lead to false positives in ARF (i.e., `whitelistAppTemporarily`) stems from not being able to resolve the caller's identity that is used to enforce a permission. The imprecision started with the authors assuming that the call from the deputy API to the target API is done over Binder IPC, which would cause the permission check in the target API to be done against the deputy's identity, while the runtime behaviour showed that the permission is still enforced against the app's identity (i.e., no permission re-delegation). Inspection of the code showed that although the call from the deputy to the target followed the general IPC pattern, it is actually handled locally as both deputy and target share the same process.

Complexity of Access Control in Android. Existing works, including DYNAMO, have focused on permissions and other security checks *in the middleware*, and we have seen how different checks interact with each other, e.g., in our analysis of inconsistencies in access control (see Section V-C). However, access control on Android also extends to the kernel using Linux DAC and SELinux MAC. For instance, although a service might seemingly be unprotected in the application framework, SELinux might prevent calls to this service from unprivileged processes. First approaches based on static analysis to model a holistic view of the access control on Android exist [32], but considering the complexity of such holistic models and the inherent limitations of the analyses from which they retrieve their data, we argue that adding dynamic analysis helps in refining such models of Android's access control.

We also observed that not all reported permissions are required for accessing an API. For example, some APIs check for permissions that are only used to select an execution path or to populate the result with more data. We identify those permission checks when no security exception is fired, and we found three patterns of how they are used: (1) The scope of the result is changed or reduced. (2) A mock or a default value is returned. (3) The execution aborts silently if the invoked API has a `void` return type. Those patterns complicate both static and dynamic analysis since the effect of permissions can only be judged from side-effects and path constraints.

Improving Static Analysis. Similar to how our design of DYNAMO profited from prior experience of static analysis (e.g., path sensitivity or security checks beyond permissions), our results can profit future static analysis. First, we found that the order of checks and their placement in the method is useful information to our inconsistency analysis. For example, we were able to easily exclude candidate paths leading to the same sink when two paths enforced similar checks *before* calling the sink. Without the location and order of the checks, we would have to manually inspect the code to see if one path is enforcing the check after calling the sink (i.e., differs in access control for the sink). In fact, not being able to argue about the order of the checks was a limitation of ARF [29]. Second, resolving the identity that is used in checks is generally a limitation of static solutions. In our permission mapping, we annotated each check with the identity used in it, including permissions enforced *after* clearing the caller's identity. This gives insights into which functionality of a service is guarded by the caller's privileges and which functionality is executed as deputy on the caller's behalf. Lack of such information (e.g.,

lack of data flow analysis) can create a faulty understanding of how system services interact with each other. Third, compromises for following IPC calls, such as limiting the analysis to first entry point [28] or improperly connecting calls [12], [5], reduced the completeness of the results. Despite current efforts [10], bridging IPC is still one of the unsolved challenges in static analysis of the application framework.

Alternative Approaches. Modeling the security policy of the application framework is a non-trivial task. The difficulty stems from the sheer size of the framework (aggravated by the closed-source modifications by OEMs), the different protection mechanisms that need to be combined in a comprehensive model (e.g., permissions, cross-user checks, etc.), and layers (Java and native code). Modeling this policy via manual reverse-engineering is not scalable and shown to be error-prone even for experienced security analysts (Section V-B). Approaches for automatic placement of authorization hooks, as known for the Linux kernel [54], [53], [60], [19], are also not feasible as they require a clear knowledge of what constitutes a protected resource, which is not available in Android's application framework despite prior efforts [12], [47]. As such, we argue that static and dynamic analysis are currently the best candidates of the available approaches to study the framework. Nevertheless, we advise symbolic execution [41] as a possible addition to address use-cases that are challenging to dynamic analysis, such as analyzing event-driven security checks that cannot be triggered dynamically because of unsatisfied state.

Code Coverage. We found the inherent code coverage problem of dynamic analysis not to pose a blocker for building permission mappings that are as complete as mappings built using static analysis. This is attributed to the fact that security checks take precedent to sanity checks and logic, and hence do not necessitate high code coverage. However, for future use-cases that need higher code coverage, our prototypical implementation has great potential to be extended, e.g., guidance from static analysis or symbolic execution results.

Limitations. Only after we conducted our evaluation, we found that more APIs could have been discovered by resolving multi-level services [38]. Moreover, deriving the security policy from the invocation results is incomplete. We manually verified over 300 APIs out of 1,292 reported APIs for Android 6 and found some APIs with security checks that were overlooked by the association rules encoded in the *Policy Builder*. Since manual verification of all APIs is out of scope, the exact FN rate is currently unclear. Moreover, dynamic testing is generally time-consuming, and DYNAMO is not different (e.g., building permission mappings took multiple weeks on our setup). Lastly, DYNAMO requires the tested device to be rooted or Android being built in debugging mode.

VII. CONCLUSIONS

We presented DYNAMO, our tool for analyzing the security policy of Android's application framework. It is based on well-known dynamic testing techniques for Java and native system APIs, considering path-sensitivity, order as well as the location of security checks, and additional key contextual information (e.g., the subject's identity). We applied DYNAMO to four prominent use-cases from the literature and by taking a synoptical view on the results, we were able to confirm, complement, and refute previous results. Root cause analysis of the discrepancies between prior and our results provided insights and new expert knowledge to improve both static and dynamic analysis in the future. We see further potential to apply dynamic analysis to Android's middleware, such as testing compliance of OEM images or scanning for vulnerabilities.

Acknowledgements. We like to thank our anonymous reviewers for their valuable comments. We also like to thank the authors of Arcade [5] and ARF [29] for their discussion with us regarding the root-cause analysis after we shared our results with them. We especially thank Arcade's authors for updating their mapping following our discussion. We also like to thank Oliver Schranz for his valuable feedback and insights on the early implementation of DYNAMO.

REFERENCES

[1] *Cuttlefish Virtual Android Devices*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://source.android.com/setup/create/cuttlefish

[2] *Randoop: Automatic unit test generation for Java*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://randoop.github.io/randoop/

[3] *WalaWiki: T.J. Watson Libraries for Analysis*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page

[4] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, "AceDroid: Normalizing diverse android access control checks for inconsistency detection," in *25th Annual Network and Distributed System Security Symposium (NDSS'18)*. The Internet Society, 2018.

[5] Y. Aafer, G. Tao, J. Huang, X. Zhang, and N. Li, "Precise android API protection mapping derivation and reasoning," in *25th ACM Conference on Computer and Communication Security (CCS'18)*. ACM, 2018.

[6] Y. Aafer, X. Zhang, and W. Du, "Harvesting inconsistent security configurations in custom android roms via differential analysis," in *25th USENIX Security Symposium (SEC'16)*. USENIX Association, 2016.

[7] A. Amin, A. Eldessouki, M. Magdy, N. Abdeen, H. Hindy, and I. Hegazy, "Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach," *Information*, vol. 10, 10 2019.

[8] P. Anderson, "The use and limitations of static-analysis tools to improve software quality," *CrossTalk-Journal of Defense Software Engineering*, vol. 21, 06 2008.

[9] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[10] 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," *ACM SIGPLAN Notices*, vol. 49, 06 2014.

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

[12] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *25th USENIX Security Symposium (SEC'16)*. USENIX Association, 2016.

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

[14] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *31st Annual Computer Security Applications Conference (ACSAC'15)*. ACM, 2015.

[15] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *J. Comput. Virol. Hacking Tech.*, vol. 13, no. 1, pp. 1–12, 2017.

[16] A. Dawoud and S. Bugiel, *Dynamo: Dynamic Analysis of the Android Application Framework*, 2020 (Last visited: Nov 17, 2020). [Online]. Available: https://github.com/abdawoud/Dynamo

[17] L. Deshotels, C. Carabas, J. Beichler, R. Deaconescu, and W. Enck, "Kobold: Evaluating decentralized access control for remote nsxpc methods on iOS," in *41st IEEE Symposium on Security and Privacy (S&P '18)*. IEEE, 2020.

[18] A. Desnos, G. Gueguen, and S. Bachmann, *Androguard's documentation*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://androguard.readthedocs.io/en/latest

[19] A. Edwards, T. Jaeger, and X. Zhang, "Runtime verification of authorization hook placement for the linux security modules framework," in *9th ACM Conference on Computer and Communication Security (CCS'02)*. ACM, 2002.

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

[21] H. Feng and K. G. Shin, "Understanding and defending the binder attack surface in android," in *32nd Annual Computer Security Applications Conference (ACSAC'16)*. ACM, 2016.

[22] Google, *Android Security 2017 Year In Review*, 2018 (Last visited: Sep 20, 2020). [Online]. Available: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf

[23] ——, *AOSP Android Filesystem Config*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://android.googlesource.com/platform/system/core/+/master/libcutils/include/private/android_filesystem_config.h

[24] ——, *Documentation for app developers*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://developer.android.com/docs

[25] ——, *TrustManager Class*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://android.googlesource.com/platform/frameworks/base/+/android-10.0.0_r27/core/java/android/app/trust/TrustManager.java#194

[26] ——, *Work Profiles*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://developer.android.com/work/managed-profiles

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

[28] S. A. Gorski III, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, and A. Bartel, "Acminer: Extraction and analysis of authorization checks in android's middleware," in *9th ACM Conference on Data and Application Security and Privacy (CODASPY'19)*. ACM, 2019.

[29] S. A. Gorski III and W. Enck, "ARF: identifying re-delegation vulnerabilities in android system services," in *12th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'19)*. ACM, 2019.

[30] Y. Gu, Y. Cheng, L. Ying, Y. Lu, Q. Li, and P. Su, "Exploiting android system services through bypassing service helpers," in *12th International Conference on Security and Privacy in Communication Networks (SecureComm'16)*. Springer, 2016.

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

[32] G. Hernandez, D. Tian, A. Yadav, B. Williams, and K. Butler, "BigMAC: Fine-grained policy analysis of android firmware," in *25th USENIX Security Symposium (SEC'20)*. USENIX Association, 2020.

[33] H. Huang, S. Zhu, K. Chen, and P. Liu, "From system services freezing to system server shutdown in android: All you need is a loop in an app," in *22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[34] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *28th International Symposium on Software Reliability Engineering (ISSRE'17)*. IEEE, 2017.

[35] 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 *2nd Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)*. ACM, 2012.

[36] H. Jiao, X. Li, L. Zhang, G. Xu, and Z. Feng, "Hybrid detection using permission analysis for android malware," in *10th International International Conference on Security and Privacy in Communication Networks (SecureComm '14)*, J. Tian, J. Jing, and M. Srivatsa, Eds. Springer, 2014.

[37] J. Lin, B. Liu, N. Sadeh, and J. I. Hong, "Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings," in *10th Symposium On Usable Privacy and Security (SOUPS '14)*. USENIX Association, 2014.

[38] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing android native system services via automated interface analysis," in *25th USENIX Security Symposium (SEC'20)*. USENIX Association, 2020.

[39] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. A. Zhang, N. Sadeh, Y. Agarwal, and A. Acquisti, "Follow my recommendations: A personalized privacy assistant for mobile app permissions," in *Twelfth Symposium on Usable Privacy and Security (SOUPS '16)*. USENIX Association, 2016.

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

[41] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation," in *15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*. ACM, 2017.

[42] S. Ognawala, A. Petrovska, and K. Beckers, "An exploratory survey of hybrid testing techniques involving symbolic execution and fuzzing," *CoRR*, vol. abs/1712.06843, 2017.

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

[44] S. T. Peddinti, I. Bilogrevic, N. Taft, M. Pelikan, U. Erlingsson, P. Anthonysamy, and G. Hogben, "Reducing permission requests in mobile apps," in *Internet Measurement Conference (IMC '19)*. ACM, 2019.

[45] S. T. Peddinti, N. Taft, I. Bilogrevic, and P. Anthonysamy, *Google Security Blog: Helping Developers with Permission Requests*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://security.googleblog.com/2020/02/helping-developers-with-permission.html

[46] O. A. V. Ravnås, *Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.*, 2020 (Last visited: Sep 20, 2020). [Online]. Available: https://frida.re/

[47] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *23rd Annual Network and Distributed System Security Symposium (NDSS'16)*. The Internet Society, 2016.

[48] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen, "Asking for (and about) permissions used by android apps," in *10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013.

[49] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J.-W. Chen, "Turtle guard: Helping android users apply contextual privacy preferences," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS '17)*. USENIX Association, 2017.

[50] R. Vallee-rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," *Conference of the Centre for Advanced Studies on Collaborative research (CASCON '99)*, 10 1999.

[51] T. Vidas, N. Christin, and L. Cranor, "Curbing android permission creep," in *W2SP*, 2011.

[52] Y. Wang, W. Cai, P. Lyu, and W. Shao, "A combined static and dynamic analysis approach to detect malicious browser extensions," *Security and Communication Networks*, vol. 2018, pp. 7 087 239:1–7 087 239:16, 2018.

[53] R. Watson, W. Morrison, C. Vance, and B. Feldman, "The trustedbsd MAC framework: Extensible kernel access control for freebsd 5.0," in

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference. USENIX Association, 2003.

[54] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. . Hartman, "Linux security modules: General security support for the linux kernel," in *11th USENIX Security Symposium (SEC'02)*. USENIX Association, 2002.

[55] J. Wu, S. Liu, S. Ji, M. Yang, T. Luo, Y. Wu, and Y. Wang, "Exception beyond exception: Crashing android system by trapping in "uncaught exception"," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP'17)*. IEEE, 2017.

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

[57] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[58] Young Han Choi, Byoung Jin Han, Byung Chul Bae, Hyung Geun Oh, and Ki Wook Sohn, "Toward extracting malware features for classification using static and dynamic analysis," in *8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC '12)*, 2012.

[59] L. Zhang, Z. Yang, Y. He, Z. Zhang, Z. Qian, G. Hong, Y. Zhang, and M. Yang, "Invetter: Locating insecure input validations in android services," in *25th ACM Conference on Computer and Communication Security (CCS'18)*. ACM, 2018.

[60] X. Zhang, A. Edwards, and T. Jaeger, "Using CQUAL for static analysis of authorization hook placement," in *11th USENIX Security Symposium (SEC'02)*. USENIX Association, 2002.

APPENDIX A
FALSE POSITIVES OF ARCADE

TABLE II. LIST OF AND REASONS FOR FALSELY POSITIVE APIS IN ARCADE'S [5] PERMISSION MAPPING

Over-Approximation
input_method.setImeWindowStatus
input_method.startInput
input_method.windowGainedFocus
iphonesubinfo.getDeviceId *
iphonesubinfo.getDeviceIdForPhone *
iphonesubinfo.getLine1Number *
iphonesubinfo.getLine1NumberForSubscriber *
phone.factoryReset
phone.getLine1NumberForDisplay *
textservices.getCurrentSpellChecker
textservices.getEnabledSpellCheckers
textservices.getSpellCheckerService
textservices.isSpellCheckerEnabled
Wrong Permission Enforcement (Name Resolution)
alarm.setTime
content.getIsSyncable *
content.getIsSyncableAsUser *
content.getMasterSyncAutomatically *
content.getMasterSyncAutomaticallyAsUser *
content.getSyncAutomatically *
content.getSyncAutomaticallyAsUser *
deviceidle.addPowerSaveTempWhitelistApp
deviceidle.addPowerSaveTempWhitelistAppForMms
deviceidle.addPowerSaveTempWhitelistAppForSms
input.registerTabletModeChangedListener
input.setTouchCalibrationForInputDevice
input.tryPointerSpeed
package.movePrimaryStorage
wallpaper.setWallpaperComponentChecked *
window.requestAssistScreenshot
window.screenshotApplications
Wrong Reporting of Relation Between Checks
package.movePrimaryStorage

(*) APIs that are accessible to third-party app developers.

APPENDIX B
FALSE NEGATIVES OF DYNAMO

TABLE III. EXAMPLES OF FALSE NEGATIVE APIS IN DYNAMO'S PERMISSION MAPPING

Over-Approximation
sip.close
power.updateWakeLockWorkSource
network_management.setUidCleartextNetworkPolicy
network_score.clearScores
account.copyAccountToUser
audio.disableSafeMediaVolume
netstats.getDataLayerSnapshotForUid
package.clearPackagePreferredActivities
media_session.setRemoteVolumeController
isms.sendDataForSubscriber
phone.getAllCellInfo
phone.getNeighboringCellInfo
appops.checkAudioOperation
bluetooth_manager.disable
connectivity.requestNetwork
input_method.setInputMethod
location.getLastLocation
vibrator.vibratePattern
account.addAccountAsUser
activity.moveTaskBackwards
device_policy.getRemoveWarning
fingerprint.getEnrolledFingerprints
package.addCrossProfileIntentFilter
window.disableKeyguard
notification.isSystemConditionProviderEnabled

APPENDIX C
RESULTS OF ARF

TABLE IV. CONFIRMED AND REFUTED VULNERABILITIES REPORTED BY ARF [29]

Confirmed Vulnerability
areNotificationsEnabledForPackage
cancelToast
dismissKeyguard
enqueueToast
getAuthenticatorId
getEnrolledFingerprints
getFreeBytes
getPermittedAccessibilityServicesForUser
hasNamedWallpaper
isDeviceSecure
isPackageDeviceAdminOnAnyUser
isSeparateProfileChallengeAllowed
False Positives
whitelistAppTemporarily
switchUser
startBluetoothSco
stopBluetoothSco
startBluetoothScoVirtualCall

APPENDIX D
RESULTS OF EVALUATING THE DEVELOPER'S DOCUMENTATION [24]

TABLE V. EXTENDED AND REFUTED PERMISSION ANNOTATIONS IN GOOGLE'S DEVELOPER DOCUMENTATION FOR MANAGER APIS [24]

APIs With Extended Permissions
activity.getCurrentUser
activity.switchUser
audio.registerAudioPolicy
color_display.setSaturationLevel
color_display.setSaturationLevel
connectivity.getCaptivePortalServerUrl
connectivity.setAirplaneMode
connectivity.shouldAvoidBadWifi
connectivity.startCaptivePortalApp
contexthub.disableNanoApp
contexthub.enableNanoApp
contexthub.queryNanoApps
incidentcompanion.deleteIncidentReports
incidentcompanion.getIncidentReport
incidentcompanion.getIncidentReportList
location.flushGnssBatch
location.getGnssBatchSize
location.registerGnssBatchedLocationCallback
network_score.setActiveScorer
phone.setRttCapabilitySetting
role.addOnRoleHoldersChangedListenerAsUser
telecom.getCurrentTtyMode
user.removeUser
wallpaper.clearWallpaper
wallpaper.setWallpaperComponent
wifi.getPrivilegedConfiguredNetworks
wifi.getWifiApConfiguration
wifi.setWifiApConfiguration
wifi.startScan
Incomplete Permission Reporting Complemented in Natural Language
role.addRoleHolderAsUser
role.clearRoleHoldersAsUser
role.getRoleHoldersAsUser
role.removeOnRoleHoldersChangedListenerAsUser
role.removeRoleHolderAsUser
usagestats.whitelistAppTemporarily
Wrong Permission Reporting
audio.getAudioVolumeGroups
overlay.getOverlayInfosForTarget
overlay.setEnabled
overlay.setEnabledExclusiveInCategory
phone.getSupportedRadioAccessFamily
phone.requestCellInfoUpdate
telecom.getPhoneAccountsSupportingScheme
telecom.isRinging
user.isRestrictedProfile

TABLE VI. MISSING PERMISSION ANNOTATIONS IN GOOGLE'S DEVELOPER DOCUMENTATION FOR MANAGER APIs [24]

Missing Permission From Online Documentation
audio.adjustStreamVolume
audio.getActivePlaybackConfigurations
audio.registerAudioRecordingCallback
audio.setBluetoothScoOn
audio.setMicrophoneMute
audio.setMode
audio.setRingerMode
audio.setSpeakerphoneOn
audio.setStreamVolume
connectivity.getRestrictBackgroundStatus
connectivity.reportBadNetwork
connectivity.reportNetworkConnectivity
connectivity.requestBandwidthUpdate
wifi.addOrUpdatePasspointConfiguration
wifi.createMulticastLock
wifi.createWifiLock
wifi.getConnectionInfo
wifi.getDhcpInfo
wifi.getScanResults
wifi.getWifiState
wifi.is5GHzBandSupported
wifi.isEasyConnectSupported
wifi.isEnhancedOpenSupported
wifi.isEnhancedPowerReportingSupported
wifi.isP2pSupported
wifi.isPreferredNetworkOffloadSupported
wifi.isTdlsSupported
wifi.isWifiEnabled
wifi.isWpa3SaeSupported
wifi.isWpa3SuiteBSupported
wifi.updateNetwork

APPENDIX E
PREDEFINED SEEDS IN DYNAMO

TABLE VII. THE LIST OF PREDEFINED SEEDS USED FOR INPUT GENERATION IN DYNAMO

Strings

`null`
`<EMPTY_STRING>`
<package name of `TS`>
com.android.systemui
content://user_dictionary/words
android.permission.MANAGER_USERS

Integers
-1, 0, 1, 10 `<INT_MAX>` <UID of `TS`> <PID of `TS`> <UID of `TS` from the second profile> <UID of com.android.systemui> <PID of com.android.systemui>
Byte, Character, Double, Float, Short
1 *
Boolean
true, false
android.content.[ComponentName/Intent]
Package: <package name of `TS`> Class: <class name of `TS`'s service>
android.net.Uri
content:://user_dictionary/words

(*) Accordingly casted to the corresponding type