PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems

Yu-Tsung Lee, *Penn State University;* William Enck, *North Carolina State University;* Haining Chen, *Google;* Hayawardh Vijayakumar, *Samsung Research;* Ninghui Li, *Purdue University;* Zhiyun Qian and Daimeng Wang, *UC Riverside;* Giuseppe Petracca, *Lyft;* Trent Jaeger, *Penn State University*

https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yu-tsung

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems

Yu-Tsung Lee[1], William Enck[2], Haining Chen[3], Hayawardh Vijayakumar[4],
Ninghui Li[5], Zhiyun Qian[6], Daimeng Wang[6], Giuseppe Petracca[7]*and Trent Jaeger[1]
[1]*Penn State University,* [2]*North Carolina State University,* [3]*Google,* 4 *Samsung Research*
[5] *Purdue University,* [6]*UC Riverside,* [7] *Lyft*
[1]*{yxl74,trj1}@psu.edu,* [2]*whenck@ncsu.edu,* [3]*hainingc@google.com,* [4]*h.vijayakuma@samsung.com,*
[5]*ninghui@cs.purdue.edu,* [6]*{zhiyunq, dwang030}@cs.ucr.edu,* [7]*petracca.giuseppe@gmail.com*

Abstract

Android's filesystem access control provides a foundation for system integrity. It combines mandatory (e.g., SEAndroid) and discretionary (e.g., Unix permissions) access control, protecting both the Android platform from Android/OEM services and Android/OEM services from third-party applications. However, OEMs often introduce vulnerabilities when they add market-differentiating features and fail to correctly reconfigure this complex combination of policies. In this paper, we propose the PolyScope tool to triage Android systems for vulnerabilities using their filesystem access control policies by: (1) identifying the resources that subjects are authorized to use that may be modified by their adversaries, both with and without policy manipulations, and (2) determining the attack operations on those resources that are actually available to adversaries to reveal the specific cases that need vulnerability testing. A key insight is that adversaries may exploit discretionary elements in Android access control to expand the permissions available to themselves and/or victims to launch attack operations, which we call *permission expansion*. We apply PolyScope to five Google and five OEM Android releases and find that permission expansion increases the privilege available to launch attacks, sometimes by more than 10x, but a significant fraction (about 15-20%) cannot be converted into attack operations due to other system configurations. Based on this analysis, we describe two previously unknown vulnerabilities and show how PolyScope helps OEMs triage the complex combination of access control policies down to attack operations worthy of testing.

1 Introduction

Android has become the most dominant mobile OS platform worldwide, deployed by a large number of vendors across a wide variety of form factors, including phones, tablets, and wearables [43]. With Android's increased integration into people's daily lives, Android needs to provide sufficient and appropriate assurances of platform integrity. Additionally, vendors must be able to extend the Android platform to support their custom functionality and yet maintain such assurances to their customers. Android's implementation of filesystem access control is one of the most important defenses for providing such assurances.

Despite aggressively adopting advanced mandatory access control (MAC) methods (e.g., SEAndroid [42]) in combination with traditional discretionary access control (DAC) for filesystem access control, Android continues to report filesystem vulnerabilities. One recent case reported by Checkpoint [30] shows how an untrusted application can abuse write permission to Android's external storage to maliciously replace a victim application's library files before it installs them, which is an example of a *file squatting attack*. Additionally, a report from IOActive [27] shows how a vulnerability in the DownloadProvider allowed untrusted applications to read/write unauthorized files by providing a maliciously crafted URI that causes DownloadProvider to access an untrusted symbolic link that redirects the victim to the targeted files, which is an example of a *link traversal attack*. This vulnerability could have a serious negative impact, as some over-the-air update files, including for some privileged applications, are downloaded by the DownloadProvider.

Researchers have proposed automated policy analysis to detect misconfigurations that may expose vulnerabilities in complex access control policies [24, 40], but application of these methods to Android does not address: (1) how those policies may be altered by adversaries and (2) how to detect which operations adversaries may actually be able to use in attacks on those misconfigurations. The emergence of Android with its rich permission system and its subsequent adoption of the SEAndroid mandatory access control motivated the development of policy analysis methods for Android systems [14, 53, 54, 1]. However, each of these initial approaches only consider a single type of access control policy (e.g., either SEAndroid or Android permissions). Recent work has computed the information flows of combined MAC and DAC policies [10], as well as including Linux capabilities [22].

*Giuseppe Petracca's work on this paper was performed when he was a graduate student at Penn State.

However, these techniques miss some attacks (including the Checkpoint [30] and IOActive [27] attacks), because they lack the ability to capture how adversaries may broaden their privilege by manipulating the inherent flexibility in the Unix and Android permission systems. These techniques may also identify many spurious threats, because they do not determine if adversaries can really launch attacks for the threats found.

In this paper, we develop a novel method called PolyScope to triage Android systems for vulnerability testing using their filesystem access control policies by: (1) identifying the resources applications are authorized to use that are also modifiable by their adversaries, both with and without policy manipulations, and (2) determining the attack operations on those resources that are available to adversaries in order to identify the specific cases that need testing for vulnerabilities. Similar to prior work [24, 40, 25, 11], our method begins by identifying filesystem resources at risk by computing the integrity violations authorized by the policy. An *integrity violation* occurs when an access control policy authorizes a lower-integrity subject (adversary) to modify a resource used by a higher-integrity subject (victim). However, an integrity violation alone does not imply a vulnerability because: (1) victims may not actually use the impacted resource and/or (2) adversaries may not be able to exploit the victim's use of the resource. Predicting which resources a program may use in advance is a difficult challenge, so instead PolyScope computes whether and how an adversary could attempt to exploit a victim's use. For each integrity violation, PolyScope computes the ways that adversaries may launch attacks, which we call *attack operations*.

A key insight of our work is that adversaries may exploit discretionary elements in Android access control to expand the permissions available to themselves and/or victims to launch attack operations, which we call *permission expansion*. DAC protection systems allow resource owners to grant permissions to their resources arbitrarily, making prediction of whether an unsafe permission may be granted intractable in general [21]. Adversaries can leverage such flexibility to grant victims permissions to lure them to resources to which adversaries can launch attack operations. In addition, Android systems often convert Android permissions that adversaries may obtain into DAC permissions, leading to further risks. While SEAndroid MAC policies bound such permissions, these MAC policies are sufficiently coarse that changes in DAC permissions may reveal many new attack operations within the MAC restrictions.

Our evaluation demonstrates that PolyScope has several benefits over prior analysis approaches and is practical to use. First, in a study of nine freshly installed Android releases,[1] five Google Android versions and four OEM Android versions, we find that permission expansion increases the number of integrity violations significantly, from 122% to 1550% across versions. However, between 14% and 21% of those integrity violations cannot be transformed into attack operations by the filesystem and/or program configurations. We also examine two Android releases spanning the introduction of the Android scoped storage defense [19], which controls use of shared external storage more tightly, showing how changes in enforcement mechanisms affect PolyScope. Second, PolyScope finds that OEM releases have a significantly greater number of attack operations than the Google releases. Using these attack operations uncovered by PolyScope, we find two new vulnerabilities in three OEM releases through manual analysis. One of these new vulnerabilities requires permission expansion to exploit, demonstrating the power of PolyScope. Finally, we implement an analysis method in PolyScope that enables parallel computation of integrity violations, resulting in significant performance improvements for the studied systems, ranging from 75% to 81% improvement across releases. This suggests that OEMs may benefit from applying PolyScope incrementally to identify attack operations as they extend their systems with new value-added features.

PolyScope is targeted for use by Android system vendors, including many OEMs, who often extend base Android systems with a variety of vendor-specific services and applications to customize their products with value-added functionality. To demonstrate its utility, we provide two case studies that show how policy misconfiguration caused by vendor-added functionality results in previously unknown vulnerabilities. These case studies show how vendors can compute attack operations that third-party applications may launch against their functionality and identify cases requiring vulnerability testing. Using PolyScope, vendors can further identify how their vendor-specific services and applications may be abused to compromise the system's trusted computing base.

This paper makes the following contributions:

- *We propose the PolyScope analysis tool for Android filesystem access control.* PolyScope composes Android's access control policies and relevant system configurations to compute the attack operations available to adversaries, accounting for permission expansion.
- *We use PolyScope to triage three Google and five OEM Android releases.* We find a significantly greater number of attack operations for OEM's Android releases, indicating that OEMs may greatly benefit from PolyScope as they customize their Android-based products.
- *We identify two new vulnerabilities in OEM Android releases.* Using PolyScope results, we identify vulnerabilities in: (1) the Thememanager used by Xiaomi and Huawei and (2) Samsung's Resestreason logging.

The remainder of this paper is as follows. Section 2 motivates the need for more effective access control analysis. Section 3 overviews of the PolyScope's approach. Section 4

[1] We examine a tenth system in some experiments, which has a significantly greater number of pre-installed apps.

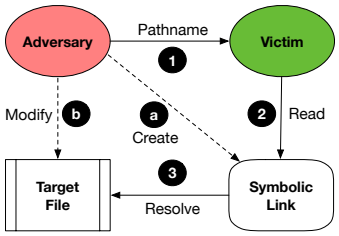

Figure 1: **DownloadProvider Vulnerability**: (1) Adversary provides pathname to victim (as URI) to (2) lure the victim to an adversary-created symbolic link (a) that (3) the victim resolves to the target file enabling the adversary to modify the file indirectly through the victim (b).

defines our threat model. Sections 5 and 6 describe the design and implementation of PolyScope. Section 7 performs a variety of experiments to show how PolyScope triages access control policies in Android releases. Section 8 describes current limitations and how they may be addressed. Section 9 examines differences from related work. Section 10 concludes.

2 Motivation

In this section, we motivate the goals of our work. We first present an example of the challenge of detecting the attack operations that may lead to vulnerabilities in Android systems in Section 2.1. After outlining current approaches to access control policy analysis in Section 2.2, we describe their key limitations in Section 2.3.

2.1 An Example Vulnerability

A recent vulnerability discovered in Android services using the DownloadProvider allows untrusted apps to gain access to privileged files [27]. The DownloadProvider enables services to retrieve files on behalf of apps by a URI specifying the location of a file. An untrusted app may lure a service's DownloadProvider into using a maliciously crafted URI that resolves to a symbolic link created by the untrusted app. Through this symbolic link, the untrusted app can access any file to which the service is authorized, which may include some privileged files. This is an example of a *link traversal attack*.

Figure 1 shows exploitation of the vulnerability. The adversary sends a request URI (Pathname in Figure 1) to the victim (service running DownloadProvider) (1) that directs the victim to a symbolic link created by the adversary (a). When the victim uses its read permission to the symbolic link (2), the operating system resolves the link (3) to return access to the target file. This vulnerability may enable the adversary to leak and/or modify the target file (b) to which the adversary normally lacks access.

This vulnerability occurred because adversaries of the service have the permission to create a symbolic link in a directory to which the service running DownloadProvider also has access. Android access control aims to limit the exposure of services to directories modifiable by third-party apps and other adversaries, but sometimes functional requirements demand such permissions be available. In addition, Android systems allow apps to extend their own permissions by obtaining Android permissions and grant services permissions to app directories, expanding the directories at risk.

2.2 Access Control Policy Analysis

To prevent such vulnerabilities, defenders may limit access to sensitive resources using access control or other techniques that "sandbox" untrusted programs. However, privileged processes often provide functionality that requires shared access to some sensitive resources with untrusted processes. In the example above, the Download Provider provides a file access service for its clients, so both share access to files served. As a result, we need a way to triage access control policies to identify resources that may lead to vulnerabilities for such authorized sharing.

Access control policy analysis [24, 40] computes the authorized information flows among subjects and objects from a system's access control policies. An access control policy authorizes an *information flow from a subject to an object* if the policy allows that subject to perform an operation that modifies the object, called a *write-like operation*, and authorizes an *information flow from an object to a subject* if the policy allows that subject to perform an operation that uses the object's data, called a *read-like operation* (e.g., read or execute). Some operations may be both read-like and write-like, enabling information flow in both directions.

However, modern Android systems have hundreds of thousands of access control rules, so there are many, many authorized information flows. Researchers then explored methods to find the information flows associated with potential security problems. Some access control analyses focus on identifying secrecy problems [10, 14, 53, 54, 1] and others on integrity problems [25, 11]. In the example above, this vulnerability permits attacks on process integrity by controlling the file retrieved by the victim process, whereby the compromised process may be directed to leak or modify files on behalf of the adversary. To detect integrity problems, access control analyses are inspired by integrity models, such as Biba integrity [4], to detect information flows from adversaries to victims. Such information flows are called *integrity violations*, which are defined more formally as a tuple of resource, adversary, and victim, where the access control policy authorizes an information flow from the adversary to the resource (i.e., the adversary is authorized to perform a write-like operation on the resource) and authorizes an information flow from the resource to the victim (i.e., the victim is authorized to perform a read-like operation on the resource).

2.3 Limitations of Current Techniques

Access control policy analyses attempt to solve three main problems to help identify vulnerabilities, but current ap-

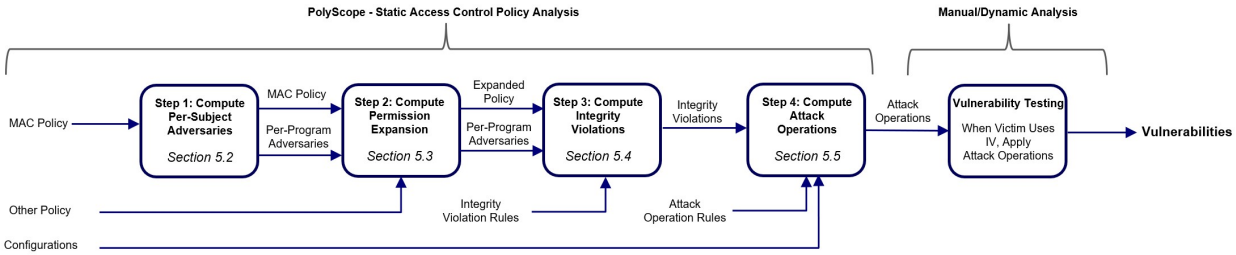

Figure 2: **PolyScope Logical Flow:** PolyScope computes per-subject adversaries (Step 1), permission expansion by those adversaries (Step 2), the integrity violations to which adversaries are authorized (Step 3), and the attack operations adversaries may perform to launch attacks (Step 4) as test cases for vulnerability testing.

proaches suffer from key limitations on each problem.

The first problem in using access control policy analysis is to **identify the adversaries who may benefit from exploitation of each subject**. Previous research often identifies untrusted apps[2] as adversaries and assumes that all system services and OEM value-added apps and services are trusted. However, as OEMs push more functionality into their own Android distributions, they deploy a variety of new and modified apps and services whose trustworthiness may vary. A recent study [16] reveals that some OEM pre-installed code lacks in end-to-end quality control and might even leverage code from third parties, resulting in backdoors and other vulnerabilities [37]. By ignoring OEM apps and services, we risk missing attacks that utilize them as stepping stones to exploit Android system services. However, we must be careful not to overapproximate adversaries, which leads to false positives.

The second problem is to **determine the permissions adversaries control to create integrity violations**. Recent access control analysis methods that reason about multiple access control policies [10, 22] do not account for how an adversary may exploit flexibility in those policy models to expose new integrity violations. For example, the recent BigMAC system computes the data flows authorized by a combination of Android policies, but we find that Android authorizes hundreds of thousands of data flows when only a small fraction (0.1 to 1.5%) cause integrity violations [3]. Another problem is that OEMs often uilitize DAC policies to protect their value-added apps and services, but adversaries may modify DAC policies to create new integrity violations by obtaining Android permissions from unsuspecting users and by granting permissions to objects they "own" to potential victims to lure them into attacks. Researchers have previously identified problems caused by DAC policy flexibility [21, 29] that limit its ability to prevent unauthorized access. While in theory MAC policies could be configured to block changes from creating integrity violations, MAC policies are more complex to configure and are unforgiving if a needed permission is not granted. As a result, we believe that OEMs are overly dependent on DAC policies.

The third problem is to **compute the operations that an adversary may be authorized to employ to launch attacks**, which we call *attack operations*. Once we know that an adversary has been authorized permissions that create an integrity violation, a question is how an adversary may exploit those permissions to launch attacks. While integrity violations are a necessary precondition for attacks, adversaries must be able to perform the operations necessary to launch attacks. Android systems provide filesystem and program configurations that could prevent attack operations. First, Android uses filesystem configurations that prevent symbolic links from being created in external storage directories, which can block link traversal attacks like the example above. In addition, Android systems have also introduced a specialized FileProvider class that requires clients to open files for their servers, which can also prevent link traversal attacks. However, such ad hoc configurations are only employed sporadically (see also Section 8.2), so it is important to determine which attack operations are really possible to guide defenders without creating false positives.

3 PolyScope Overview

In this paper, we present a new Android access control analysis tool, called PolyScope, that computes the set of authorized attack operations for an Android system while overcoming the limitations described above. Prior research that performs multi-policy analysis only computes information flows for a current snapshot of the policies [10, 22], which neither accounts for policy manipulations nor identifies the specific attack operations that could enable exploitation. Figure 2 shows PolyScope's approach. In Step 1, PolyScope identifies the adversaries for each subject using definitions of mutual trust validated against an approach that computes worst-case, as described in Section 5.2. In Step 2, PolyScope determines the permissions adversaries control by modeling how adversaries may expand permissions available to themselves and their victims by exploiting the flexibility in Android access control policies, as described in Section 5.3. In Step 3, PolyScope uses

[2] Includes apps assigned to the SEAndroid domains `untrusted_app` and `isolated_app`. Information on SEAndroid domains appears in Section 5.1.

[3] Compare the "Authorized Data Flows" row in Table 2 to the "PolyScope IVs with Operations" row.

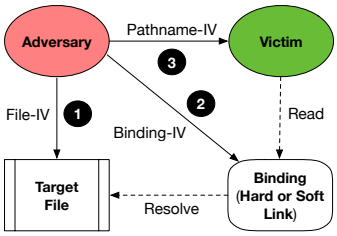

Figure 3: **Integrity Violation (IV) Classes**: (1) *File-IVs* grant adversaries direct access to modify files that victims use; (2) *Binding-IVs* grant adversaries the ability to modify name resolution of file names; and (3) *Pathname-IVs* enable adversaries to lure victims to the part of the filesystem they can modify.

these expanded permissions to compute integrity violations based on *integrity violation rules* defined in Section 5.4. In Step 4, PolyScope uses these integrity violations to compute the types of attack operations possible using *attack operation rules* defined in Section 5.5. We identify the specific types of integrity violations and attack operations we consider in this paper in Section 4.

PolyScope computes integrity violations and attack operations to triage Android releases for vulnerabilities authorized by access control policies. Integrity violations computed in Step 3 identify the filesystem resources that victims are authorized to access that their adversaries are authorized to modify (see Section 2.2). Attack operations computed in Step 4 determine the types of operations that adversaries are capable of performing in modifying filesystem resources to launch attacks for each IV. As indicated in Figure 2, all the PolyScope steps perform static access policy analysis. Using the computed attack operations, an analyst can perform vulnerability testing on victim applications either manually or preferably using dynamic analysis. The aim is to develop dynamic testing techniques that drive victims to scenarios where they access a resource associated with an integrity violation, where testing will apply an attack operation to determine whether the victim prevents the attack or not. In Section 6, we describe dynamic analysis analysis methods to detect victim use of IVs from which we find two new vulnerabilities from subsequent manual testing in Section 7.6.

4 Threat Model

In this paper, adversaries may modify any part of the filesystem and send requests (e.g., IPCs) to any subject to which they are authorized by the combination of Android access control policies. Adversaries may make arbitrary changes to authorized filesystem resources. Also, adversaries may send arbitrary data in requests. We assume that adversaries will exploit such abilities to modify any resource that they are authorized to modify that a victim may use. That is, adversaries we assume that adversaries will exploit all integrity violations (IVs) authorized by the Android policies.

We aim to triage systems for three classes of integrity violations on filesystem access, covering a wide variety of vulnerabilities including confused deputy [20] and time-of-check-to-time-of-use (TOCTTOU) vulnerabilities [31, 5]. Related to Figure 1, we show these integrity violation classes in Figure 3. First, *file-IVs* allow adversaries to modify target files that are authorized to victims directly (1), possibly leading victims to unexpected use of adversary-controlled data. We further distinguish file-IVs by whether the victim can read (*read-IVs*), write (*write-IVs*), and/or execute (*exec-IVs*) the IV file. Second, *binding-IVs* enable adversaries to redirect victims to target files during name resolution (2), causing victims to operate on files chosen by adversaries. Third, *pathname-IVs* enable adversaries to lure victims to an adversary-controlled part of the filesystem using an adversary-supplied pathname (3), which is the integrity violation exploited in the example vulnerability of Section 2.1.

For each integrity violation found, we assume that an adversary may attempt any possible attack operation. In general, attack operations enable adversaries to provide malicious input to victims by getting them to use an adversary-controlled file or binding to enable the adversary to choose the input to the victim, whose basic approaches date to the 1970s [31]. File-IV attack operations simply *modify the resource* awaiting use (read, write, or execute) by the victim. Binding-IV attack operations direct the victim to a resource chosen by the adversary, using link traversal or file squatting attacks. A *link traversal* attack directs a victim to access a resource to which the adversary is not authorized. A *file squatting* attack plants an adversary-controlled resource at a location where the victim expects a protected file. Pathname-IVs attack operations lure a victim who processes pathnames (e.g., URLs) to an adversary-controlled binding to exploit a link traversal.

In developing PolyScope, we assume trust in some components of Android systems. First, we assume that the Android operating system operates correctly, including enforcement of its access control policies and system configurations correctly. For example, we trust the Android operating system to satisfy the reference monitor concept [2]. We note that the Android operating system includes the Linux operating system and a variety of system services. Our assumptions about trust among such services is determined using Android specifications, as described in Section 5.2.

5 PolyScope Design

In this section, we examine the design challenges in computing attack operations for Android systems. In particular, after providing some brief background information, we focus on four key steps outlined in the PolyScope overview in Section 3.

5.1 Design Background

In this section, we describe the various access control techniques used by Android systems that are necessary to un-

derstand the PolyScope design. Android uses SEAndroid mandatory access control (MAC), Unix discretionary access control (DAC), the Android permission system, and Linux capabilities to control access to filesystem resources directly or indirectly. Linux capabilities have no tangible impact on contributing attack operations on recent Android versions, so we do not discuss them further. Using the remaining models, we define PolyScope's interpretation of *subjects* and *objects* applied in policy analysis.

SEAndroid MAC: SEAndroid is a port of the SELinux mandatory access control system [39] with additional support for Android mechanisms, such as Binder IPC. SEAndroid supports three access control models: Type Enforcement (TE), Role-Based Access Control (RBAC), and Multi-Level Security (MLS). All the models are *mandatory access control models* (MAC) in that they are defined by the system and are not intended to be modified by users or their programs. Out of these three, the TE model acts as the primary enforcement model to protect the integrity of the Android system's trusted computing base processes. MLS is used mainly to separate apps from one another. On the other hand, RBAC does not receive much use currently on Android, so we do not describe it further.

The SEAndroid TE policy[4] defines authorization rules in terms of security labels [6], where a subject can perform an operation on an object if there is a rule authorizing the subject's security label to perform the operation on object's security label. The SEAndroid MLS policy enables subjects and objects to be associated with categories [3], where subjects can only perform operations on objects when the subject is authorized for the object's category.

Unix DAC: Android systems also use traditional Unix discretionary access control (DAC) as provided by the Linux system on which Android is based. Unix DAC associates files with a UID for the file owner and a GID for the file group. Processes are also associated with a process UID and GID, but a process may additionally belong to a set of supplementary groups. A process can perform an operation on a file if: (1) the file's UID is the same as the process's UID and the file owner is authorized to perform that operation; (2) the file's GID matches one of the process's groups (i.e., process's GID or supplementary) and the file group is authorized to perform that operation; or (3) any process UID (i.e., others) is authorized to perform that operation for that file. Importantly, Unix DAC allows a process to modify file permissions when the process's UID is the same as the file's owner UID.

Rather than associating UID's with individual users, as is traditional, Android associates UIDs with individual services and apps. Thus, services and apps "own" a set of files (i.e., with the app's UID as the file owner UID) for which they may modify permissions. Thus, malicious apps can change the permissions for files they own, which is important for luring victims to create pathname-IVs.

Android Permission System: Android permissions are used to control access to app and service data. Android data/service providers enforce most Android permissions, but some Android permissions are mapped to DAC supplementary groups, which are assigned to apps when the associated Android permission is granted. Thus, Android permissions may add DAC supplementary groups to app processes, granting them additional filesystem permissions.

Note that each Android permission has an associated *protection level* that is used to determine whether or not an application may be granted that permission. Over time, the permission granting policy has become more complex [57]. Currently, permissions with the "normal" protection level are automatically granted to applications. However, permissions with the "dangerous" protection level (e.g., guarding sensitive personal data such as GPS) require additional runtime authorization from the user. Permissions with the "signature" protection level can only be granted to applications signed by the same developer key that was used to define the permission. The signature protection level is primarily used to restrict access to functionality that only system applications should access. Finally, there are several other flags that provide ad hoc restrictions, e.g., a "privileged" flag allows privileged, OEM-bundled applications to acquire associated permissions.

Mapping MAC and DAC Policies: To reason about access control for the combined DAC and MAC policies, PolyScope needs to determine how to map MAC policies in the form of TE security labels and MLS categories to DAC policies in the form of UIDs and groups. Fortunately, Android makes such determination straightforward [5]. Files and directories are explicitly assigned both MAC and DAC information directly, so there is no possibility of ambiguity. For processes, the mapping between MAC and DAC information is indirect. Android assigns the same MAC TE security label, MAC MLS category, DAC UID, and DAC groups[6] to each program when it is run, as identified by Chen et al. [10]. Thus, we collect the MAC-DAC mapping for processes by running programs. For all apps and services we have run, this relationship has held, but this implies that we can only perform policy analysis for apps and services installed on the release (i.e., that we can run). We use this information to define subjects and objects for PolyScope analysis as follows.

- **Subjects**: Each process is associated with a *subject* defined by its MAC TE label, MLS category set, DAC UID, and a set of DAC groups (GID and supplemental

[4]Note that in this paper we sometimes refer to the "MAC TE" policy, which is the same as the SEAndroid TE policy.

[5]Extracting such policy information is straightforward when we have the filesystem information. We root Android devices to obtain such information, but researchers have developed techniques to extract such information from firmware images [22], as we discuss in Section 6.

[6]The complete set of supplementary groups assigned to a program's processes depend on the Android permissions obtained for the program. We define the assumption we use for PolyScope in Section 5.3.

groups). There may be many processes associated with one subject.

- **Objects**: Each resource is associated with an *object* defined by its MAC TE label, MLS category set, DAC UID/GID, and mode bits (i.e., owner, group, others permissions). There may be many files/directories associated with one object.

PolyScope reasons about access control policies in terms of subjects and objects, rather than individual processes and resources, as the definitions of subjects and objects form equivalence classes with respect to adversaries. All processes of the same subject share the same adversaries, and all resource associated with the same object are modifiable by the same adversaries. Thus, we express PolyScope results in terms of subjects and objects in Section 7.

5.2 Compute Per-Subject Adversaries

One challenge is to identify the adversaries for each subject. Researchers often identify subjects adversaries based untrusted sources (e.g., Chen et al. [10] used third-party apps as adversaries) or based on their role in the system (e.g., Jaeger et al. [25] said only core system services could be trusted). However, these approaches are one-dimensional and ad hoc. Since they are ad hoc, there is a greater likelihood of missing possible attack sources or identifying trusted sources as false adversaries. Since they are one-dimensional (i.e., either focusing on trust or not), there is no basis to determine whether adversaries are missing or misclassified.

We propose to develop a method for computing adversaries that considers both the best-case and worst-case trust to derive per-subject adversary sets. For the worst case, we leverage the conservative *integrity wall* approach of Vijayakumar *et al.* [50]. The integrity wall approach uses the insight that only the subjects that can trivially compromise a subject's program must be trusted, thus computing a minimal trusted computing base (TCB) (i.e., fewest subjects trusted) of subjects for each program. For the best case, we use the process privilege levels defined by Google [17], which groups subjects in classes by whether they should be mutually trustworthy. By examining trust from two directions, we can perform validation on whether the combination is consistent. While this approach enables just one type of limited validation, we are not aware of any prior work validating adversary sets.

The integrity wall method computes per-subject TCBs by detecting whether either one of two requirements are met: (1) the subject must trust any other subjects that are authorized to modify files that the subject may execute (i.e., its executable and library files) and (2) the subject must trust any other subjects that are authorized to modify kernel resources. We compute worst-case per-subject TCBs from MAC TE policies.

On the other hand, Android specifies "privilege levels" [17] that describe which subjects should mutually trust one another, implying a best-case TCB. Google defines six privilege levels in an Android system [17], which we group into five levels for evaluation in Section 7: (T5) root processes; (T4) system processes; (T3) service processes; (T2) trusted application processes; (T1) untrusted application processes and isolated process. Isolated apps and untrusted apps are separated into distinct privilege levels by Google, but in this paper, we do not consider attacks on untrusted apps by the lower privileged isolated apps. Table 1 lists these privilege levels based on their UID and MAC labels.

Using Google's privilege levels, we assume a subject trusts all of the subjects at its level or higher. For example, untrusted apps trust other untrusted apps and any subjects at higher privilege levels, such as the Android system services (e.g., system server). Trusting subjects at higher privilege levels is accepted because such subjects provide functionality that the subjects at lower privilege levels depend upon. However, assuming untrusted apps may be mutually trusting may be harder to accept, but we are not looking for attacks between untrusted apps in this paper. Resolving how to identify adversaries among untrusted apps, such as determining whether mutual trust for all is appropriate, is future work.

To produce an accurate adversary set, we validate consistency between the best-case and worst-case trust sets to derive an adversary set that is consistent with both trust sets. Specifically, PolyScope validates whether the worst-case trust set for each subject is a subset of that subject's best-case trust set. If so, then there does not exist an adversary of any subject relative to its best-case trust set (i.e., fewest adversaries) that is not also an adversary relative to that subject's worst-case trust set (i.e., maximal adversaries). An inconsistency implies that the Android privilege levels are missing a fundamental trust requirement to prevent trivially compromising the subject.

5.3 Compute Permission Expansion

A key difficulty for OEMs is predicting which resources may be accessible to adversaries and victims to derive attack operations accurately. A problem is that while MAC policies are essentially fixed (i.e., between software updates), DAC permissions may be modified by adversaries to increase the attack operations that they could execute. We identify two ways that adversaries may modify permission assignments on Android systems: (1) adversaries may obtain Android permissions that augment their own DAC permissions and (2) adversaries may delegate DAC permissions for objects that they own to potential victims. For some Android permissions, adversaries gain new DAC permissions to access additional resources that may enable attacks. By delegating DAC permissions to objects they own, adversaries may lure potential victims to resources that adversaries control.

Adversary Permission Expansion: In Android systems, some Android permissions are implemented using DAC groups. As described above, a process is associated with a single UID and GID, but also an arbitrarily large set of supplementary groups that enable further "group" permissions. Thus,

Table 1: Google's Process Privilege Levels [17]

Process Level[1]	Level Membership Requirements
Root Process (T5)	Process running with UID root (e.g., MAC labels `kernel` and `init`)
System Process (T4)	Process running with UID system (e.g., MAC label `system server`)
Service Process (T3)	AOSP core service providers (e.g., MAC labels `bluetooth` and `mediaserver`)
Trusted Application Process (T2)	AOSP default and vendor apps (e.g., MAC labels `platform_app` and `priv_app`)
Untrusted Application Process (T1)	Third-party applications (e.g., MAC label `untrusted_app`)
Isolated Process (T0)	Processes that are expected to receive adversarial inputs (e.g., MAC label `webview`)

[1] Listing types of processes based on their privilege level, from high to low with root processes being most privileged (T5) and isolated processes being the least privileged (T0). We group T0 and T1 together calling the resultant level T1 in the evaluation in Section 7.

when a user grants an Android permission associated with one or more DAC groups to an app, there is a direct expansion of that app's permissions in terms of its DAC permissions. Since the MAC policies are generally lenient in Android systems, these new DAC permissions may grant privileges that enable attacks. For PolyScope, we assume that subjects can obtain all of their "normal" Android permissions and are granted all of their "dangerous" permissions by users for analysis, as described in the previous section. One of the vulnerability case studies we highlight in Section 7.6 exploits adversary permission expansion.

Victim Permission Expansion: Researchers have long known that allowing adversaries to administer DAC permissions for their own objects can present difficulties in predicting possible permission assignments. Researchers proved that the *safety problem* of predicting whether a particular unsafe permission will ever be granted to a particular subject for a typical DAC protection system, i.e., an access matrix for subjects and objects where objects and permissions may be added in a single command, is undecidable in the general case [21]. As a result, researchers explored alternative DAC models for which the safety problem could be solved, such as the take-grant model [26], the typed access matrix [38], and policy constraints [45].

Using the ability to manage DAC permissions to objects they own, adversaries can grant permissions to their resources to victims, expanding the set of resources that victims may be lured to use. In many cases, victims have MAC permissions that grant them access to adversary directories, but vendors use DAC permissions to block access. However, since adversaries own these directories, they can simply grant the removed permissions to potential victims themselves.

5.4 Compute Integrity Violations

In this section, we show how to compute integrity violations for file-IVs, binding-IVs, and pathname-IVs defined in Section 4. Recall from Section 2.2 that integrity violations are a tuple of resource, adversary, and victim, where the adversary is authorized to modify the resource and the victim is authorized to use (e.g., read, write, or execute) the resource.

Computing File Integrity Violations: A file vulnerability may be possible if a subject uses (read, write, or execute) a file that can be modified by an adversary. In practice, many subjects read files their adversaries may write (`read-IVs`) with adequate defenses, but risks are greater if the subject executes (`exec-IVs`) or also modifies such files (`write-IVs`). For exec-IVs, executing input from an adversary enables an adversary to control a victim's executable code. For write-IVs, if a subject writes to a file its adversaries also may write, then adversaries may be able to undo or replace valid content.

```
{read|write|exec}(file, victim) &&  // victim can access file,
adv-expand(file, adversary) &&      // but adv-expanded perms
write(file, adversary)              // enables to modify file
    -->
{read|write|exec}-IV(file, victim, adversary)
```

This rule determines whether the victim is authorized by the combination of access control policies for reading, writing, or executing files, using the `{read|write|exec}` predicate. The rule accounts for the adversary's expansion of their own permissions, as indicated by the predicate `adv-expand`. If the adversary also has write permission to the file (`write` predicate), then the associated integrity violation is created.

Computing Binding Integrity Violations: A binding vulnerability is possible if a subject may use a binding in resolving a file name that adversaries can modify.

```
use(binding, victim) &&          // victim can use binding,
adv-expand(file, adversary) &&   // but adv-expanded perms
write(binding, adversary)        // enable to modify binding
    -->
binding-IV(binding, victim, adversary)
```

This rule parallels the rule for file-IVs, except that this rule applies to a victim having the permission to use a binding in name resolution (`use` predicate).

Computing Pathname Integrity Violations: Pathname integrity violations are binding integrity violations that are possible when a subject uses input from an adversary to build a file pathnames used in name resolution. First, adversaries must be authorized to communicate with the victim. Second, through their input, adversaries can lure victims to any bindings they choose, enabling them to expand the IVs available for exploitation by victim permission expansion.

```
write(ipc, adv, vic) &&          // may send IPCs to victim
vic-expand(binding, adv, vic) && // and expand victim perms
binding-IV(binding, vic, adv)    // to lure victim
    -->
pathname-IV(binding, vic, adv)
```

Adversaries must be granted write privilege to communicate to the victim, as defined in the `write` predicate. Android services may use Binder IPCs, and we further limit `write`

to use IPCs that communicate URLs for Android services. The adversary can use victim expansion to increase the set of bindings the victim is authorized to use by `vic-expand`. If that binding meets the requirements of a binding-IV (see above), then a pathname-IV is also possible for this victim.

5.5 Compute Attack Operations

We define how PolyScope computes attack operations from the integrity violations computed in the last section and the relevant system configurations. We identify four types of attack operations that an adversary could use to exploit the three types of integrity violations: (1) file modification for file IVs; (2) file squatting for binding-IVs; (3) link traversal for binding-IVs; and (4) luring traversal for pathname-IVs.

File Modification Attacks: For read/write/exec IVs, the attack operation is to modify the objects involved in each IV. However, Android uses some read-only filesystems, so not all files to which adversaries have write privilege are really modifiable. Thus, the rule for *file modification* operations additionally checks whether the file is in a writable filesystem.

```
{read|write|exec}-IV(file, victim, adversary) &&
fs-writable(file)      // file's filesystem is writable
    -->
file-mod(file, victim, adversary)
```

File Squatting Attack: In a file squatting attack, adversaries plant files that they expect that the victim will access. The adversary grants access to the victim to allow the victim to use the adversary-controlled file. This attack operation gives the adversary control of the content of a file that the victim will use. To perform a file squatting attack operation, the adversary must really be able to write to the directory to plant the file. Thus, the rule for *file squatting* operations is essentially the same as for file modification, but applies to binding-IVs.

```
binding-IV(binding, victim, adversary) &&
fs-writable(binding)     // binding's filesystem is writable
    -->
file-squat(binding, victim, adversary)
```

In this rule, we assume that the adversary predicts the filenames used by the victim. In the future, we will explore extending the rule to account for that capability.

Link Traversal: A link traversal attack is implemented by planting a symbolic link at a binding modifiable by the adversary, as described in Section 2.1. However, Android also uses some filesystem configurations that prohibit symbolic links, so not all bindings to which adversaries have write privilege and are in writable filesystems allow the creation of the symbolic links necessary to perform link traversals. Thus, the rule for *link traversal* operations extends the rule for file squatting to account for this additional requirement.

```
binding-IV(binding, victim, adversary) &&
fs-writable(binding)     // binding's filesystem is writable
symlink(binding) &&      // and allows symlinks
    -->
link-traversal(binding, victim, adversary)
```

Luring Traversal: An adversary may lure a victim to a binding controlled by the adversary to launch an attack operation. However, the Android FileProvider class can prevent such attacks. Specifically, the FileProvider class requires that clients open files themselves and provide the FileProvider with the resultant file descriptor. Since the clients open the file, they perform any name resolution, so the potential victim is no longer prone to pathname vulnerabilities. Thus, the rule for *luring traversal* operations extends the rule for link traversal for pathname-IVs by requiring the absence of any FileProvider class usage. OEMs still have many services and privileged apps that do not employ the FileProvider class, leaving opportunities for pathname-IVs to be attacked.

```
pathname-IV(binding, victim, adversary) &&
fs-writable(binding) &&    // binding's filesystem is writable
symlink(binding) &&        // and allows symlinks
no-file-provider(victim)   // victim does not use FileProvider
    -->
luring-traversal(binding, file, victim, adversary)
```

While it is possible that the victim has implemented an extra defense in Android middleware (e.g., Customized Android Permission) to prevent IPCs, we do not yet account for these defenses. Including these defenses is future work.

6 Implementation

The PolyScope tool is implemented fully in Python in about 1500 SLOC and is compatible with Android version 5.0 and above. After *Data Collection* gathers access control policies, PolyScope implements the logical flow shown in Figure 2 in a slightly different manner described below. First, PolyScope computes integrity violations in steps one to three in Figure 2, but only for the SEAndroid TE policy, which we call *TE IV Computation*. Next, PolyScope computes whether the TE integrity violations are authorized by the remaining Android access control policies by re-running steps two and three in Figure 2, but only for resources associated with the TE IVs, which we call *TE IV Validation*. This separation enables us to parallelize the validation step, which has a significant performance impact, see Section 7.7. Finally, PolyScope leverages the validated IVs to *Compute Attack Operations*. Below, we discuss these major phases of the implementation, and how we use the results in *Testing for Vulnerabilities*.

Data Collection: We implemented multiple data collection scripts that collect access control data for the subjects and objects from an Android phone. The methods are relatively straightforward for accessible files and processes, detailed in Appendix A.1. However, we are not authorized to access all files, particularly those owned by root, so we run these scripts on rooted phones. Recent work by Hernandez et al. [22] is able to extract MAC policy and part of DAC configuration from Android firmware images without rooting devices. However, it has difficulty obtaining files located in some directories like `/data`. As shown in Table 1 of their paper [22], about 75% of the files' DAC configuration in `/data` cannot be retrieved, which we extract with our scripts.

TE IV Computation: To compute per-subject adversaries in Step 1 of Figure 2, PolyScope leverages the integrity wall [50] and Android privilege levels [17], as described in Section 5.2. We follow the procedure defined in the integrity wall paper for Linux [50], except we add objects upon which Android kernel integrity depends (e.g., `rootfs` and `selinuxfs`) to the set of kernel objects. Since the SEAndroid TE policy is immutable (i.e., modulo system upgrades), Step 2 of Figure 2 is not required. In Step 3, PolyScope computes the integrity violations authorized by the TE policy, as specified in Section 5.4.

TE IV Validation: After the TE IVs are identified, PolyScope validates whether these TE IVs are also authorized for the combination of remaining Android access control policies: Unix DAC, SEAndroid MLS, and Android permissions. Step 1 of Figure 2 is not rerun. In Step 2, PolyScope converts Android permissions to authorized DAC subgroups for adversary expansion and identifies the objects owned by each subject for victim expansion, as described in Section 5.3. In Step 3, PolyScope determines whether the SEAndroid MLS and DAC policies also authorize the victim and adversary of each IVs. As mentioned above, the set of TE IVs can be partitioned to validate them in parallel.

Compute Attack Operations: PolyScope then computes the attack operations for the IVs using the filesystem and program configurations as described in Section 5.5. PolyScope collects the relevant filesystem configurations by parsing the associated mount options. PolyScope collects the relevant program configurations (i.e., whether the victim includes a recommended defense, the FileProvider class) by reverse engineering the application's `apk` package to detect the presence of the FileProvider class. We validated the ability or inability to perform attack operations and found no discrepancies.

Testing for Vulnerabilities: The ultimate goal is to determine whether the victim is vulnerable to any of the attack operations. However, a key challenge is to determine whether and when a victim may actually access a resource associated with an attack operation. Just because a potential victim may be authorized to use a resource, does not mean it ever uses that resource. Even if a potential victim may use a resource associated with an attack operation, we need to determine the conditions when such an access is performed. Thus, detecting vulnerabilities often requires runtime testing.

The major challenge is to drive the victim subjects' programs to cause all file usage operations, akin to fuzz testing. Developing a fuzz testing approach for file operations is outside the scope of this paper, so we simply drive programs with available tools: (1) Android Exerciser Monkey; (2) Compatibility Testing Suite (CTS); and (3) Chizpurfle [23]. We use the Android Exerciser Monkey and CTS to emulate normal phone usage, and Chizpurfle to drive Android system services. With this approach, we are able to find the vulnerabilities described in Section 7.6. We discuss how to employ runtime systematically in the future in Section 8.1.

7 Evaluation

Table 2 summarizes the highlights of our evaluation for nine fresh installs of Android releases, demonstrating the importance of computing per-victim adversaries, permission expansion, and attack operations. Table 2 shows the relative effort to vet Android releases for vulnerabilities using the output of prior analyses [22, 10] (*Authorized Data Flows*), output of a past analyses [25, 50] using PolyScope's method for computing adversaries (*IVs for PolyScope Adversaries*), and two new analyses performed by PolyScope (*PolyScope IVs after Expansion* and *PolyScope IVs with Operations*) that provide a more accurate accounting of the threats victims may face. The counts are shown in terms of subject-object pairs, as *subjects* and *objects* are defined in Section 5.1. For data flows, we sum of the objects that each subject is authorized to use (i.e., in a *read-like operation*, see Section 2.2). For integrity violations (IVs), we only count the data flows to objects that another subject classified as an adversary is authorized to modify (i.e., in a *write-like operation*, see Section 2.2).

The first row of Table 2 lists the number of *authorized data flows* allowed by Android access control policies, showing that analyses that only compute data flows [22, 10] leave OEMs with hundreds of thousands of cases to assess to detect vulnerabilities. The second row in Table 2 shows that the number of data flows to consider can be reduced significantly by only considering those that cause integrity violations. The way PolyScope computes adversaries per-victim (see Section 5.2) for finding the *IVs for PolyScope Adversaries* results in a reduction of the number of data flows involved in integrity violations by at least two orders of magnitude.

Additionally, PolyScope provides two new analysis steps to detect threats more accurately. First, the third row of Table 2 shows the number of *PolyScope IVs after expansion*, which shows the counts for IVs found using the rules defined in Section 5.4. In several cases, the number of integrity violations increases significantly after accounting for expansion, in some cases by more than 10x. Second, the fourth row of Table 2 shows that the number of *PolyScope IVs with operations* based on the rules in Section 5.5 may be significantly reduced (14-21% across these releases) because no attack operation may be possible for some IVs given the filesystem and/or victim subjects' program configurations. We also consider the impact of the Android scoped storage defense [19] applied to Android 11 on PolyScope. Table 2 shows that number of IVs increases greatly between Android 10 and 11. As discussed in Section 8.2, we find that the traditional access controls were weakened when the new defense was added [7]. The total number of attack operations possible is shown in the fifth row, indicating the effort to test each release for vulnerabilities in terms of the types of operations that must be tested.

In Sections 7.1 to 7.4, we examine how the PolyScope

[7] Android scoped storage was introduced as an option in Android 10 and made mandatory in Android 11.

Table 2: Summary of Impact of PolyScope Analyses

	Google Devices					OEM Devices			
	Nexus 5x 7.0	**Nexus 5x 8.0**	**Pixel3a 9.0**	**Pixel3a 10.0**	**Pixel3a 11.0**	**Mate9 8.0**	**Mate9 9.0**	**Mix2 9.0**	**Galaxy S8 9.0**
Authorized Data Flows[1]	204,241	166,027	156,315	161,689	169,884	240,916	860,508	289,238	259,992
IVs for PolyScope Adversaries[2]	167	80	69	71	264	223	166	192	628
PolyScope IVs after Expansion[3]	372	478	1,139	1,059	2,127	1,682	1,566	2,304	4,377
PolyScope IVs with Operations[4]	297	387	927	898 [6]	1,764[7]	1,331	1,327	1,979	3,736
Total Attack Operations[5]	350	417	962	997	1,999[7]	2,160	1,777	2,137	5,063

Unit is the relation {Subject, Object}, where subjects and objects are defined in Section 5.1
[1] Objects authorized for use by Subjects
[2] Authorized Data Flows where Object is modifiable by at least one PolyScope per-victim adversary, see Section 5.2
[3] PolyScope Integrity Violations (sum for all types) as defined in Section 5.4
[4] PolyScope Integrity Violations in at least one Attack Operation, see Section 5.5
[5] Sum of Attack Operations (in Table 4)
[6] Assumes opting out of Android scoped storage, see Section 8.2
[7] Does not account for cases blocked by Android scoped storage, see Section 8.2

implementation (see Section 6) impacts the analysts' efforts to vet their releases for vulnerabilities. In Section 7.5, we assess the distribution of IVs across privilege levels. In Section 7.6, we describe how we found two types of previously unknown vulnerabilities using PolyScope output. Finally, we measure PolyScope's analysis performance in Section 7.7.

7.1 TE IV Computation

RQ1: *How many integrity violations are found when using the SEAndroid MAC TE policy alone in TE IV Computation?* PolyScope's implementation computes IVs initially using only the SEAndroid MAC TE policy. Android has relied heavily on MAC TE to protect important daemons and system services since its introduction in Android 5.0, as shown by the number of **MAC TE allow rules**[8] in Table 3. Due to its immutable nature, the MAC TE policy provides a foundation for Android access control that other policies can modify.

The three **TE** rows (rows 2-4) of Table 3 show the number of binding-IVs, write-IVs, and read-IVs for the MAC TE policy using the rules in Section 5.4. We note that in counting the TE IVs, we only use the MAC TE labels to identify subjects and objects, which results in coarser-grained subjects and objects than Section 5.1. Thus, the TE IV counts presented represent a lower bound. We found this sufficient for the qualitative comparison with IV counts after TE IV validation below. The pathname-IV count is not shown as it is the same as the binding-IV count, as the TE policy produces no additional pathname-IVs because permission expansion is not allowed for the MAC TE policy.

7.2 TE IV Validation

RQ2: *How are the number of integrity violations (IVs) reduced after TE IV Validation from those found in the TE IV Computation?* The next four rows (rows 5-8) in Table 3 show the number of IVs for the four IV types in Section 5.4 after considering TE IV Validation (**Valid**) using other Android access control policies[9]. We see that the number of TE IVs (rows 2-4) is much greater than the number of valid IVs (rows 5-8), even accounting for the coarser subjects and objects applied in the TE IV counts[10].

Recall in Table 2 that the total IV counts after permission expansion are much higher across every release, showing that more testing to detect vulnerabilities is required than just testing IVs from the current policy. However, we observed that the SEAndroid MLS policy does effectively prevent several opportunities for victim permission expansion for objects in application-private directories (e.g., `/data/data`). If MLS can be effectively applied to Android filesystems more broadly that may greatly reduce the opportunities for victim permission expansion.

7.3 IVs for OEM Customizations

RQ3: *How do OEM customizations impact the Android integrity violation counts across vendors?* To make their products more attractive, OEMs customize Android images to provide vendor-specific, value-added functionality and more attractive user interfaces. We are interested to see how OEM customization affects the number of integrity violations created when the OEMs have to customize their Android access control policies. The devices of choice are as follows: Huawei Mate9 on Android O and Android P, Xiaomi Mix2 on Android O and Android P, and Samsung Galaxy S8 on Android P. The results are shown in the right half of Table 3.

We can see heavy customization of the MAC policy. Every OEM has a significantly greater number of MAC allow rules than the Google MAC policies in the left half of Table 3. This suggests OEMs have introduced many new domains for their own services and apps, and granted them a wide variety

[8] The drastic increase of MAC allow rules from Android 7 to 9 can be largely attributed to the effect of Google's Project Treble [18] in Android 8, which introduced many new MAC domains due to the decomposition of the Hardware Abstraction Layer (HAL). TE rules leading to the use of scoped storage (see Section 8.2).

[9] Note that the total IV count shown in Table 2 for *PolyScope IVs after Expansion* row is equal to the sum of *Valid Read-IVs* and *Valid Pathname-IVs* rows in Table 3. The Write-IVs are a subset of the Read-IVs (i.e., all victims have read access to IVs when they have write access) and the Binding-IVs are a subset of the Pathname-IVs (i.e., victims can still access binding-IVs through luring).

[10] In addition, 25% of TE IVs cannot be validated because the MAC-to-DAC mapping for some subjects is not known, see Section 8.1. Although this is a large number of TE IVs, the combination of policies still reduces the Valid IV counts much more significantly.

Table 3: Integrity Violations across Vendor Releases

	Google Devices					OEM Devices				
	Nexus 5x 7.0	**Nexus 5x 8.0**	**Pixel3a 9.0**	**Pixel3a 10.0**	**Pixel3a 11.0**	**Mate9 8.0**	**Mate9 9.0**	**Mix2 8.0**[1]	**Mix2 9.0**	**Galaxy S8 9.0**
MAC TE allow rules[2]	64,830	133,545	191,556	38,845	43,902	250,220	276,181	273,295	282,650	498,941
TE Write-IVs[4]	468	411	1,130	1,513	1,342	2,067	1,958	1,657	2,197	1,787
TE Read-IVs[4]	1,410	2,373	4,296	3,940	3,369	8,922	9,890	8,370	8,423	10,912
TE Binding-IVs[3]	495	438	693	705	513	1,504	1,233	1,400	1,174	1,881
Valid Write-IVs[4]	120	19	56	63	913	400	236	232	216	469
Valid Read-IVs[4]	194	80	85	87	1,014	679	437	531	749	953
Valid Binding-IVs[3]	52	22	32	37	190	217	159	248	154	550
Valid Pathname-IVs[3]	178	398	1,054	972	1,113	1,003	1,129	1,186	1,555	3,424
Valid IVs Total[5]	372	478	1,139	1,059	2,127	1,682	1,566	1,717	2,304	4,377

TE implies only having permission in SEAndroid TE
[1] This phone has significantly more files perhaps related to a higher number of pre-installed apps
[2] Unit: number of rules
[3] Unit: IVs (victim, object) for directory objects only
[4] Unit: IVs (victim, object) for file objects only
[5] Valid IVs Total is the same as the *PolyScope IVs after Expansion* from Table 2: the sum of the Valid Read-IVs (includes all Valid Write-IVs) and Valid Pathname-IVs (includes all Valid Binding-IVs)

of MAC permissions. The result of this customization is a significant increase MAC TE integrity violations, often more than twice as many as the associated Google Android systems. Even more importantly, the number of integrity violations is significantly higher for the OEMs after TE IV validation (rows 5-8 in Table 3). For example, the number of binding-IVs in Android version 9.0 systems is 32 for Google and at least 154 for the OEM Android 9.0 systems.

7.4 IVs to Attack Operations

RQ4: *How many attack operations are really possible for the IVs found across OEM releases?* Table 2 shows that not all the IVs found after permission expansion enable adversaries to launch attack operations because filesystem and/or victim subjects' program configurations may prevent attack operations, as described in Section 5.5.

Table 4 breaks down how many attack operations of each type are possible given the configurations that may block such operations. The number of *file attack* operations (adversary writes) are roughly the same as the number of read integrity violations (**Valid Read-IVs**), because not many objects associated with integrity violations reside in read-only directories. The number of *file squat attack* operations is the same as the number of integrity violations for directories (**Valid Binding-IVs**) in Table 3. However, the number of *link traversal attack* operations that are possible is fewer than the number of integrity violations because not all filesystems support symbolic links, reducing the number of directories where this attack operation applies.

The *luring traversal attack* operations row identifies the number of luring traversal attacks that could be performed via Binder IPC, see Section 5.5. We can easily see that the number of operations is a lot greater than the number of binding-IVs alone (**Valid Binding-IVs**), since adversaries can expand the victim's permissions for pathname-IVs (**Valid Pathname-IVs**). This is especially the case for Android 11, but this is addressed via the scoped storage defenses [19] discussed in Section 8.2. Recall that FileProvider usage is key to preventing luring traversal attacks (see the `luring-traversal` rule in Section 5.5), where it has a non-trivial but modest impact on reducing attack operations (14-21% across all releases). For example, on Samsung Galaxy S8, we found that 57 out of 356 Java applications utilize FileProvider for file sharing, which meant that 3,424 pathname-IVs were only reduced to 2,874 luring-traversal operations.

7.5 Cross-Privilege Level IVs

RQ5: *How are integrity violations distributed across Android privilege levels?* The IV distribution is important because it indicates how victims at each privilege level could be attacked and how adversaries at any privilege level could compose attacks to reach other privilege levels. Table 5 shows the counts of file and binding integrity violations between each pair of privilege levels we evaluated. We do not include pathname-IVs in this table to assess attack paths without luring.

Google's 8.0 and 9.0 releases have a modest number cross-privilege level IVs. This confirms our hypothesis that Google's access control policies are the closest to best practice. The Android 11.0 again depends on scoped storage to remove its IVs, as discussed in Section 8.2. However, on the OEM side, it can be a completely different story. Other than the Mate9 9.0, the IVs between each privilege level pair can be significant, meaning that even without luring, releases may be vulnerable in a variety of ways.

7.6 Vulnerability Case Studies

RQ6: *What kind of vulnerabilities may be discovered from attack operations?* Using the attack operations computed by PolyScope, we manually identified two previously unknown vulnerabilities.

Samsung Resetreason: We found a new binding vulnerability in the Samsung Galaxy S8 system using the Android 9.0 release. Samsung includes a privileged service called *resetreason* that logs the reason why the phone has had to reset into the file `power_off_reset_reason.txt` in the directory `/data/log`. However, any process that runs with the `AID_LOG` group has write permission to that file, so such processes can replace the file with a symbolic link to any file accessible to *resetreason* to launch a link traversal attack. While only signed apps may be granted the Android permission

Table 4: Attack Operations

	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Pixel3a 10.0	Pixel3a 11.0	Mate9 8.0	Mate9 9.0	Mix2 8.0[1]	Mix2 9.0	Galaxy S8 9.0
File Attack[2]	176	70	79	103	864	597	358	478	655	862
Link Traversal Attack[3]	1	8	3	2	2	169	7	175	4	507
File Squat Attack[3]	52	22	32	37	190	660	443	248	154	847
Pathname Attack[3]	121	317	848	892	943	734	969	761	1,324	2,874
Total Attack Operations	350	417	962	997	1,999	2,160	1,777	1,662	2,137	5,063

Unit: Sum of operations for all (victim, object) IVs
[1] This phone has significantly more files perhaps related to a higher number of pre-installed apps
[2] Only for file objects
[3] Only for directory objects

Table 5: Cross-Privilege Level IVs

	Nexus 5x 7.0	Nexus 5x 8.0	Pixel3a 9.0	Pixel3a 10.0	Pixel3a 11.0	Mate9 8.0	Mate9 9.0	Mix2 8.0*	Mix2 9.0	Galaxy S8 9.0
T1* $\rightarrow$ **T2**[1][2]	28	6	17	24	239[3]	54	29	124	24	64
T1 $\rightarrow$ **T3**	40	22	21	29	342	17	12	40	25	22
T1 $\rightarrow$ **T4**	30	13	7	11	58	14	8	29	14	12
T1 $\rightarrow$ **T5**	24	9	6	7	28	16	8	23	8	12
T2 $\rightarrow$ **T3**	40	22	21	29	342	20	15	60	48	92
T2 $\rightarrow$ **T4**	30	13	7	11	58	14	8	78	72	199
T2 $\rightarrow$ **T5**	24	9	6	7	28	20	11	34	16	41
T3 $\rightarrow$ **T4**	31	24	16	19	63	265	129	85	87	124
T3 $\rightarrow$ **T5**	68	28	14	15	22	108	126	42	107	46
T4 $\rightarrow$ **T5**	0	0	0	0	0	0	0	0	0	0

* T1(untrusted/isolated app), T2(priv/platform app) T3(services), T4(system app, system service), T5(root service)
[1] For adversary at lower level (T1) and victim at higher level (T2)
[2] Unit: Sum of binding and file IVs (no pathname-IVs included)
[3] Much higher due to weaker DAC defense

(`READ_LOGS`) associated with the `AID_LOG` DAC group, vendors include several signed apps on their devices, and some signed apps have had reported vulnerabilities, such as the `adb` app [33]. `resetreason` has access to several integrity-critical resources, and we have confirmed that we can redirect `resetreason` to write files in the encrypted filesystem directory. Previous work demonstrated the importance of attacks on the encrypted filesystem from the system's radio service [44]. We responsbly reported this vulnerability, which has been confirmed by Samsung and assigned CVE-2020-13833.

Xiaomi and Huawei Thememanager: We discovered multiple unreported vulnerabilities in the Xiaomi and Huawei devices. We describe one example here. These devices include a variety of value-added services, including the Thememanager, which allows users to customize the user interface of their devices. However, the Xiaomi access control policies are configured such that untrusted apps can write to the file `/data/data/com.android.thememanager/cache`, which is used by the Thememanager for storing content that the Thememanager may use in configuring the display. We verified on Xiaomi 8.0 that arbitrary modifications to this file do crash the privileged Thememanager process and in some cases impact the GUI without crashing. A finely-crafted modification could perhaps exploit the Thememanager service. We found four other similar vulnerabilities in the Xiaomi 8.0 release for writeable cache files. We responsibly reported these vulnerabilities to Xiaomi, who indicated that they were fixed in the Xiaomi 9.0 release.

We found that Huawei on both the 8.0 and 9.0 releases has a similar vulnerability for the theme cache files as well, but exploitation requires adversaries to compromise an application with `media_rw` permission (T2 in the Google Privilege Levels) We responsibly reported these vulnerabilities to Huawei who stated that they are not concerned about so-called privileged apps being exploited. We note that a similar scenario led to the issuing of a CVE by Samsung. Furthermore, we point out that privileged applications have been found to be flawed in several instances, see Section 2.3.

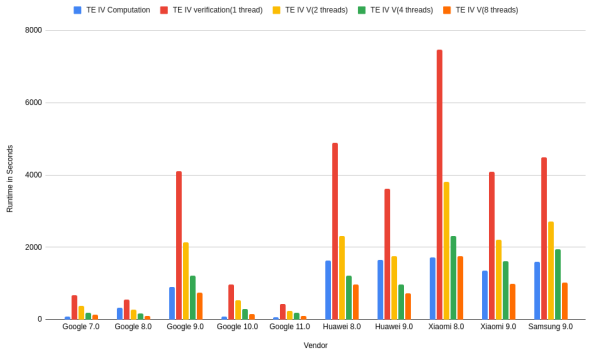

Figure 4: PolyScope Analysis Performance

7.7 Performance

We measured the performance of PolyScope for the eight Android releases. The overhead was measured on a PC running an AMD Ryzen 7 3700X (8 core, 16 thread) with 16GB of RAM and an RTX 2080 Super GPU using Ubuntu 18.04. PolyScope IVs are found in two steps as described in Section 6: TE IV computation and TE IV validation. We find that the performance of TE IV computation has a linear relationship to the SEAndroid policy size. The TE IV validation stage's performance is proportional to the number of IVs found in TE IV computation, but that impact can be reduced because validation can be parallelized.

Figure 4 shows the performance overhead of these two stages[11] for the eight releases. We evaluate the performance

[11] The cost of computing attack operations is negligible and included in the TE IV validation.

of the TE IV validation for one to eight threads. With a multi-core CPU, parallelization does produce significant performance improvement. We also point out that we found it quite expensive to compute all the authorized data flows for these releases. On the other hand, with a proper threat model to prune cases, PolyScope is able to identify integrity violations in a reasonable amount of time.

8 Discussion

In this section, we review limitations in the PolyScope approach and examine the implications of a recently proposed Android defense called *scoped storage*.

8.1 Limitations

We identify three limitations of PolyScope: (1) PolyScope relies on rooted phone to collect filesystem data; (2) we cannot always determine the mapping between MAC labels and their corresponding DAC UIDs; (3) PolyScope cannot confirm vulnerabilities from attack operations automatically.

Without rooting the phone, we cannot gather DAC information from privileged directories such as `/system`. Recently, Hernandez et al. [22] proposed BigMAC, which includes a technique to extract accurate DAC configuration data from these privileged directories ($\tilde{9}5\%$). We will explore integrating BigMAC into our data collection in future releases. Together with the data collected from an unrooted phone, see Appendix A.1, we should be able to recover a nearly complete snapshot of the filesystem. We will explore methods to achieve complete recovery in future work.

Another limitation of PolyScope is that finding the MAC-to-DAC mapping of subjects requires running a process for each MAC label to collect its DAC UIDs/groups.[12] Currently, if either the adversary or victim for a computed TE IV is not mapped to a complete subject, we skipped the IV validation stage for that TE IV. About 25% of the TE IVs do not go through validation. Runtime support could collect such mappings to seed validation.

Finally, PolyScope lacks a systematic way to test the victims for vulnerabilities to the attack operations found. The problem is that we need to know when a victim uses a file, binding, or IPC that is associated with an attack operation. Sting [52] provides passive runtime monitoring of processes for use of bindings associated with attack operations (e.g., file squatting and link traversal). However, Sting only used the available DAC policies to determine whether an attack operation would be possible, and did not test for other attack operations. PolyScope's more accurate computation of attack operations should improve the effectiveness of such an approach. To test for luring traversals, one must develop methods to detect at-risk IPCs rather than file accesses. The Jigsaw system [49] provides a method for identifying system calls that may receive input that could enable luring traversals, but it does not identify the scope of targets to which luring may occur. PolyScope identifies a full scope of luring targets using victim permission expansion, so we will explore the use of PolyScope to generate test cases for the system calls identified by Jigsaw.

[12] Recall that we leverage the finding of Chen *et al.* [10] that the MAC-to-DAC mapping for Android systems is one-to-one.

8.2 Scoped Storage

Android *scoped storage* [19] was recently introduced to control application access to another's files in the external storage folders (e.g., Download) that are shared among applications. For these shared folders, scoped storage limits app accesses only to the files they create, except for apps with the READ_EXTERNAL_STORAGE[13] Android permission. Even in this case, apps cannot modify files they did not create. For private folders, scoped storage prevents an app from reading another app's files. These restrictions prevent use of victim permission expansion (see Section 5.3) to create pathname-IVs and prevent exploitation of many attack operations on other file-IVs.

These defenses are enforced by *filesystem in userspace* (FUSE), which has been re-introduced in Android 11. When a file operation is issued to external storage, the permission checking is done at the FUSE-daemon, which leverages the MediaProvider's database to keep track of file ownership. Scoped storage was deployed as an option in Android 10 (e.g., apps may opt-out), but now is mandatory in Android 11.

Scoped storage impacts PolyScope by preventing many of the IVs found in external storage folders from being used in attack operations. In Android 11, MAC and DAC permissions have been weakened to grant apps access to files in shared external storage folders, so PolyScope identifies these as IVs. Table 5 shows that Android 11 has many IVs between privilege levels T1 and T3. These weakened policies also create risks among apps at the same privilege level, but we do not consider that threat in this paper. However, scoped storage prevents attack operations from being exercised on many of these IVs in external storage, excepting only read-IVs in shared directories. We estimate that the number of *PolyScope IVs with Operations* in Table 2 is reduced from 1,764 by about half for Android 11 due to scoped storage. While this indicates a large number of false positives, actually only 30 objects are misclassified. Since Table 2 counts the subject-object pairs, the weakened MAC and DAC policies that grant several subjects access to these objects, which exacerbates the impact. At present, scoped storage is only applied in external storage folders, so scoped storage would not block attack operations on the vulnerable IVs identified in Section 7.6.

Extending PolyScope to reason about scoped storage is future work. We have two obvious choices for including

[13] MANAGE_EXTERNAL_STORAGE provides read/write access to files on external storage, but is now a signature-level permission that requires Google's approval. At the time of writing, Google halted the granting of this signature permission due to the workload of app vetting.

scoped storage. First, PolyScope could be extended to analyze policies enforced by the FUSE-daemon for scoped storage analogously to MAC and DAC policies. PolyScope could be extended to analyze policies enforced by the FUSE-daemon for scoped storage analogously to MAC and DAC policies by extending our model of subjects and objects in Section 5.1 for the policies managed by the MediaProvider. Second, PolyScope could be extended to reason about the scoped storage enforcement semantics at large, e.g., by preventing any attack operation in external storage folders. This approach would be simpler in concept, but one would need to ensure that PolyScope always had the correct semantics for scoped storage as the system evolves. Since some external storage folders used for gaming are not processed by the FUSE-daemon (i.e., are outside scoped storage), tracking folders accurately could be non-trivial. We will explore implementing the first option.

9 Related Work

Researchers have long known about the three types of integrity violations listed in Section 4, but have found it difficult to prevent programs from falling victim to such threats. A variety of mechanisms have been proposed to prevent attacks during name resolution, including defenses for binding and pathname vulnerabilities. These defenses have often been focused on TOCTTOU attacks [31, 5]. Some defenses are implemented in the program or as library extensions [12, 35, 13, 46] and some as kernel extensions [28, 36, 9, 34, 47, 48], but the methods overlap, where some enforce invariants on file access [12, 28, 48, 35, 36, 47], some enforce namespace invariants [9, 34], and some aim for "safe" access methods [13, 46]. In general, all program defenses have been limited because they lack insight into the changing system and all system defenses are limited because they lack side-information about the intent of the program [8].

The main defense for preventing filesystem vulnerabilities is access control. If the access control policies prevent an adversary from accessing the filesystem resources that enable attack operations, then the system is free of associated vulnerabilities. However, the discretionary access control (DAC) policies commonly used do not enable prediction of whether a subject may obtain an unauthorized permission [21], so techniques to restrict DAC [26, 38, 45] and apply mandatory access control (MAC) enforcement [3, 4] were then explored, culminating in MAC enforcement systems, such as Linux Security Modules [55] employed by SELinux [39] and AppArmor [32]. Researchers than proposed MAC enforcement for Android systems [56, 7], so a version of SELinux [39] targeting Android was developed, called SEAndroid [42]. However, the attack operations we find in this paper abuse available MAC permissions. While a techniques have been developed to limit processes the permissions available to individual system calls [41, 51], such techniques need policy analysis to determine the policies to enforce.

Researchers have proposed using access control policy analysis to identify misconfigurations that may lead to vulnerabilities [24, 40], but traditionally, access control policy analysis methods only reason about one policy, such as a mandatory access control (MAC) policy [40, 25, 11, 50] or an Android permission policy [14, 53, 54]. However, based on the research challenges above, we must consider the combination of the access control policies employed on the system to compute attack operations accurately. Chen *et al.* [10] were the first work that we are aware of to combine MAC and DAC policies in access control policy analysis. Hernandez et al. [22] further extended their analysis to include MAC, DAC and Linux capabilities. However, both of these techniques compute data flows, which are much more numerous than integrity violations. Chen *et al.* look for data flows that may lead to sensitive data leakage directly rather than attack operations that may enable such leakage as PolyScope does.

10 Conclusions

Android uses a combination of filesystem access control mechanisms to assure its platform integrity. This paper has proposed PolyScope, a policy analysis tool that reasons over Android's mandatory (SEAndroid) and discretionary (Unix permissions) access control policies, in addition to the other mechanisms (e.g., Android permissions) that influence file access control. PolyScope is novel in its ability to reason about permission expansion, which lies at the intersection of mandatory and discretionary policy. We applied PolyScope to three different Google Android releases and five different OEM Android releases, characterizing the potential for file-based attacks such as file squatting, link traversal, and luring traversal. In doing so, we identified two new vulnerabilities in OEM Android releases and opportunities to direct further automated testing. Our results suggest that the access control policy changes introduced by OEMs do not sufficiently address integrity violations for their feature additions.

Acknowledgments

Thanks to our shepherd, Sven Bugiel, and the anonymous reviewers. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and National Science Foundation grants CNS-1816282. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

References

[1] Yousra Aafer, Nan Zhang, Zhongwen Zhang, Xiao Zhang, Kai Chen, XiaoFeng Wang, Xiaoyong Zhou, Wenliang Du, and Michael Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1248–1259, 2015.

[2] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical report ESD-TR-73-51, AFSC, October 1972.

[3] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), March 1976.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical report MTR-3153, MITRE, April 1977.

[5] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[7] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Network and Distributed System Security Symposium(NDSS)*, 2012.

[8] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE Statistical Signal Processing Workshop*, 2009.

[9] Suresh Chari, Shai Halevi, and Wietse Venema. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *Proceedings of the 17th Network and Distributed System Security Symposium(NDSS)*, 2010.

[10] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[11] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the 16th Network and Distributed System Security Symposium(NDSS)*, pages 11–16, 2009.

[12] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-hartman. RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, 2001.

[13] Drew Dean and Alan Hu. Fixing Races for Fun and Profit. In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[15] Boris Farber. ClassyShark. URL: https://github.com/google/android-classyshark. Accessed May 2020.

[16] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An Analysis of Pre-installed Android Software. *arXiv preprint arXiv:1905.02713*, 2019.

[17] Google. Security Overview. 2019. URL: https://source.android.com/security/overview/updates-resources#process_types. Accessed Jan. 10, 2020.

[18] Google. SELinux for Android 8.0. February 2018. URL: https://source.android.com/security/selinux/images/SELinux_Treble.pdf. (Accessed Dec 2019).

[19] Google. Storage Updates in Android 11. URL: https://developer.android.com/preview/privacy/storage. Accessed June 2020.

[20] Norm Hardy. The Confused Deputy: or Why Capabilities Might Have Been Invented. *ACM Special Interest Group in Operating Systems, Operation System Review*, 22(4), 1988. ISSN: 0163-5980.

[21] M. Harrison, W. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of ACM*, August 1976.

[22] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *Proceedings of the USENIX Security Symposium*, 2020.

[23] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, and Cristina Nita-Rotaru. Chizpurfle: A Gray-box Android Fuzzer for Vendor Service Customizations. In *Software Reliability Engineering (ISSRE), IEEE 28th International Symposium*, pages 1–11, 2017.

[24] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Managing Access Control Policies Using Access Control Spaces. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 3–12, New York, NY, USA, 2002.

[25] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, 2003.

[26] A. Jones, R. Lipton, and L. Snyder. A Linear Time Algorithm for Deciding Security. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, 1976.

[27] Daniel Kachakil. Multiple Vulnerabilities in Android's Download Provider (CVE-2018-9468, CVE-2018-9493, CVE-2018-9546). https://ioactive.com/multiple-vulnerabilities-in-androids-download-provider-cve-2018-9468-cve-2018-9493-cve-2018-9546/, January 2020.

[28] Kyung-suk Lee and Steve J. Chapin. Detection of File-based Race Conditions. *International Journal of Information Security*, 2005.

[29] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[30] Slava Makkaveev. Man-in-the-Disk:Android Apps Exposed via External Storage. February 2019. URL: https://research.checkpoint.com/2018/androids-man-in-the-disk/.

[31] W. S. McPhee. Operating System Integrity in OS/VS2. *IBM System Journal*, 13:230–252, 3, September 1974.

[32] Novell. AppArmor Linux Application Security. http://www.novell.com/linux/security/apparmor/.

[33] Open ADB Ports Being Exploited to Spread Possible Satori Variant in Android Devices, August 2018. URL: https://blog.trendmicro.com/trendlabs-security-intelligence/open-adb-ports-being-exploited-to-spread-possible-satori-variant-in-android-devices. (Accessed Feb 2020).

[34] OpenWall Project - Information Security Software for Open Environments, 2008. URL: %5Curl%7Bhttp://www.openwall.com/%7D.

[35] J. Park, G. Lee, S. Lee, and D. Kim. RPS: An Extension of Reference Monitor to Prevent Race-Attacks. In *Advances in Multimedia Information Processing*, 2004.

[36] Calton Pu and Jinpeng Wei. Modeling and Preventing TOCTTOU Vulnerabilities in Unix-style Filesystems. In *IEEE International Symposium of System Engineering*, 2006.

[37] Ryan Johnson. All Your SMS and Contacts Belong to Adups and Others. July 2017. URL: https://www.blackhat.com/docs/us-17/wednesday/us-17-Johnson-All-Your-SMS-&-Contacts-Belong-To-Adups-&-Others.pdf. (Accessed June 2019).

[38] R. S. Sandhu. The Typed Access Matrix Model. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, 1992.

[39] SELinux. -. URL: https://github.com/SELinuxProject. (Accessed Dec 2019).

[40] SETools. URL: https://github.com/TresysTechnology/setools. Accessed Dec 2019.

[41] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS)*, 2006.

[42] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the 20th Network and Distributed Systems Symposium (NDSS)*, 2013.

[43] StatCounter. OS Market Share. March 2020. URL: https://gs.statcounter.com/os-market-share.

[44] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *27th USENIX Security Symposium*, pages 273–290, 2018.

[45] Jonathon Tidswell and Trent Jaeger. An access control model for simplifying constraint expression. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000.

[46] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *USENIX Conference on File and Storage Technologies*, 2008.

[47] Eugene Tsyrklevich and Bennet Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. In *USENIX Security Symposium*, 2003.

[48] Prem Uppuluri, Uday Joshi, and Arnab Ray. Preventing Race Condition Attacks on Filesystems. In *ACM Symposium on Applied Computing*, 2005.

[49] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the 23rd USENIX Security Symposium*, August 2014.

[50] Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, and Trent Jaeger. Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 75–76, 2012.

[51] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process Firewall: Protecting Processes During Resource Access. In *Proceedings of the Eighth European Conference on Computer Systems*, 2013.

[52] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding Name Resolution Vulnerabilities in Programs. In *21st USENIX Security Symposium*, 2012.

[53] Ruowen Wang, Ahmed M. Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[54] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-scale Semi-supervised Learning. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 351–366, 2015.

[55] Chris Wright, Crispin Cowan, and James Morris. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security Symposium*, 2002.

[56] Liang Xie, Xinwen Zhang, Ashwin Chaugule, Trent Jaeger, and Sencun Zhu. Designing System-Level Defenses against Cellphone Malware. In *28th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2009.

[57] Yury Zhauniarovich and Olga Gadyatskaya. Small Changes, Big Changes: An Updated View on the Android Permission System. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

A Additional Background

In this section, we provide details on how PolyScope collects the relevant access control information.

A.1 Access Control Data Collection

MAC Data: To obtain MAC data, PolyScope first pulls the SEAndroid policy binary file from the Android root directory with command "`adb pull sepolicy`". With the SELinux policy binary in hand, we extract the allow rules with "`sesearch -A sepolicy`". Then, in order to parse the SELinux attributes, we pull the attribute mapping with "`seinfo -a -x sepolicy`".

DAC Data: To obtain DAC permissions for all files on an Android system, PolyScope executes "`adb shell ls -lRZ`" from the root directory. Note that the phone must be rooted to obtain the full list of file permissions, so we use a boot time root technique to gain root. PolyScope collects the file permission data shown in Table 6. The data in Table 6 indicates: a file `authtokcont` under the directory `/efs` has `read, write` permissions for its owner and group members. Its owner and group UID are both `radio`, and its MAC security label is `efs_file`.

Table 6: File DAC data sample

File	DAC perms	User	Group	MAC security label
`authtokcont`	`-rw-rw-r--`	`radio`	`radio`	`efs_file`

Process Information: PolyScope obtains process access control information by executing the command "`adb shell ps -A -o label,user,group,COMMAND`", which provides a mapping from a DAC user ID to MAC label for running processes. One data sample is shown in Table 7. This entry shows that process `init` has security label of `u:r:init:s0`, UID of `root`, GID of `root`, was spawned by command `/init`, and PID of `1`. However, the process list collection does not provide the full information on DAC supplementary groups, as we described in Section 5.3. In the case of Android system services, these extra groups are defined in the `init.rc` file, which can be parsed statically. For apps, PolyScope uses a shell script to obtain process DAC group information stored in `/proc`.

Table 7: Process Data

Security label	UID	Group	Command	PID
`u:r:init:s0`	`root`	`root`	`/init`	`1`

Android Permission Data: To obtain Android Permissions' mappings to DAC groups, PolyScope parses `/etc/platform.xml` from the Android device. Next, we need to separate the signature Android Permissions from the non-signature Android Permissions, which are available via the Android package manager (PM), as the non-signature permissions may be applied by an app. PolyScope uses the non-signature permissions to compute DAC expansion for adversaries.

Filesystem and FileProvider: To determine whether attack operations are blocked, PolyScope needs to examine the filesystem configuration and the application package. First, PolyScope obtains filesystem configurations by running "`adb shell mount`", which will return list of filesystem mount configuration. We identify the directories mounted with the `ro` or the `nosymlink` flags and mark them as read-only and prohibiting symlinks, respectively. Second, for the application package, we want to determine if the application uses the FileProvider class to protect itself from luring. PolyScope first queries the PackageManager service for a full list of apk files on the system. Next, PolyScope collects all the apk files found and performs code inspection with Google's new ClassyShark tool [15] to identify the presence of the FileProvider class.