PolyScope: Multi-Policy Access Control Analysis to Triage Android Scoped Storage

Yu-Tsung Lee, Haining Chen, William Enck, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Giuseppe Petracca and Trent Jaeger, *IEEE Senior Member*

Abstract—Android's filesystem access control is its foundation for system integrity. It combines mandatory (e.g., SELinux) and discretionary (e.g., Unix permissions) access control with other specialized access controls (e.g., Android permissions), aiming to protect Android/OEM services from third-party applications. However, OEMs often introduce vulnerabilities when they add market-differentiating features because they fail to correctly reconfigure this complex combination of policies. In this paper, we present the PolyScope tool, which triages the combination of Android filesystem access control policies to find the authorized operations that may be exploited by adversaries to escalate their privileges, called *attack operations*. Critically, PolyScope accounts for how adversaries may modify permissions for themselves and/or their victims to uncover latent attack operations. We demonstrate the effectiveness of PolyScope by assessing the impact of the recently introduced *Scoped Storage* defense for Android, showing that extending PolyScope to analyze a new policy can be done independently if the new policy only restricts permissions, which is the case for Scoped Storage. We apply PolyScope to three Google and five OEM Android releases, finding that Scoped Storage reduces the number of attack operations possible on external storage resources by over 50%. However, we also find two previously unknown vulnerabilities because OEMs only adopt Scoped Storage partially, limiting its benefit. Thus, we show how to use PolyScope to assess an ideal scenario where all apps are compliant to Scoped Storage, which can reduce the number of untrusted parties that can access attack operations by over 65% on OEM systems. As a result, we find that PolyScope can help Android OEMs triage complex access control policies to identify the specific attack operations worthy of further examination.

Index Terms—Access control, Access control policy analysis, Mobile security, Android security

1 Introduction

Android, the dominant mobile OS worldwide [41], powers a wide range of devices. Ensuring platform integrity and customizable functionality is crucial as Android integrates further into daily lives. Filesystem access control in Android serves as a key defense for providing such assurances to vendors and users.

Because Android allows its users to run untrusted, third-party apps, access control methods have been applied aggressively to try to protect platform integrity from these apps. Android systems enforce a combination of discretionary access control (DAC) (e.g., Unix permissions) and mandatory access control (MAC) (e.g., SEAndroid [40]) along with specialized access controls (e.g., Android permissions). Unlike DAC, which allows users and their processes to modify the permission assignments to the objects that they own, MAC enables system distributors (i.e., Google or OEMs) to define an immutable access control policy that confines processes to a fixed set of permissions, even if they are compromised. In addition, Android permissions allow users to grant permissions to apps, also providing a discretionary means for users to modify permissions[1].

However, even with SEAndroid MAC, Android continues to experience filesystem vulnerabilities. For example, Checkpoint [30] reports 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*. In addition, a vulnerability in the ContactsProvider allows untrusted applications to open/delete/insert files in unauthorized locations by providing maliciously crafted URIs [14], e.g., to an adversary-controlled symbolic link referencing an unauthorized file in an example of *link traversal attack*. Researchers have shown that these two classes of attacks are possible when an adversary is authorized to write filesystem resources used by a victim [49].

To detect filesystem vulnerabilities, researchers have proposed applying automated access control policy analysis techniques [22, 38] to Android systems [13, 50, 51, 1]. These analyses convert individual policy rules into information flows, which help identify secrecy (e.g., data leakage) and integrity problems (e.g., use of untrusted executables). Recent advances include computing information flows for combined MAC and DAC policies [7], as well as incorporating Linux capabilities [19]. However, detecting authorized information flows based on existing access control poli-

- *Yu-Tsung Lee and Trent Jaeger are with Penn State University. E-mail: yxl74@psu.edu,trj1@psu.edu*
- *Haining Chen is with Google. E-mail: hainingc@google.com*
- *William Enck is with North Carolina State University. E-mail:whenck@ncsu.edu*
- *Hayawardh Vijayakumar is with Samsung Research North America. E-mail:h.vijayakuma@samsung.com*
- *Ninghui Li is with Purdue University. E-mail:ninghui@cs.purdue.edu*
- *Zhiyun Qian is with UC Riverside. E-mail:zhiyunq@cs.ucr.edu*
- *Giuseppe Petracca's work was done when he was a student at Penn State. E-mail:petracca.giuseppe@gmail.com*

Manuscript received April 19, 2022; revised August 26, 2022.

1. For more details about MAC and DAC enforcement in Android, please refer to Section 5.1 of the paper by Lee et al. [27].

This article has been accepted for publication in IEEE Transactions on Dependable and Secure Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2023.3310402

cies alone is insufficient to detect attacks for two reasons. First, such techniques may overlook certain attacks, like the Checkpoint [30] and ContactsProvider [14] attacks, as they fail to capture how adversaries can exploit the inherent flexibility in Unix and Android permission systems to broaden their privileges as well as those of their victims. Second, these techniques may also generate numerous false positives by not assessing whether the identified information flows can genuinely be exploited to launch attacks, e.g., considering the system configurations.

To address these limitations, we developed a novel access control policy analysis tool, called PolyScope [27], that triages Android access control policies to identify the specific attack operations that adversaries are authorized to launch. In this paper, we show how the PolyScope design addresses three key issues in multi-policy access control analysis to triage recent Android 11 and 12 policies. First, the PolyScope design leverages the insight that access control policies whose decisions are combined via intersection can be analyzed independently [52], enabling the addition of new policy models without impacting others. Second, PolyScope accounts for how adversaries may exploit discretionary elements in Android access control to expand the permissions available to themselves and/or victims, which we call *permission expansion*. By accounting for permission expansion, PolyScope can detect attacks that are missed by analyses that use policies as configured. Third, PolyScope reasons about how permissions may be exploited in attacks to convert unsafe data flows computed by past systems [23] into specific *attack operations* that can really be performed.

To demonstrate the effectiveness of PolyScope to triage Android systems for filesystem vulnerabilities, we investigate the impact of the recently introduced *Scoped Storage* access control defense [16, 26]. Android uses a separate filesystem partition for many dynamically processed applications files, including media and application updates, called the *external storage partition* for historical reasons, but this filesystem has been the source of many exploits, including Checkpoint vulnerability [30]. We show how PolyScope enables the independent addition of the Scoped Storage policy model to the existing combination of Android access control models to assess research questions about the impact of this defense using eight freshly installed Android releases: three Google Android versions and five OEM Android versions. PolyScope shows that Scoped Storage reduces the number of possible attack operations in external storage across Google and OEM systems by over 50%. However, on OEM devices, PolyScope shows that a small number of "legacy apps" that are not compliant with Scoped Storage are authorized for many attack operations, even two previously unknown vulnerabilities. We show how to use PolyScope to evaluate a what-if scenario where all apps are compliant with Scoped Storage, finding that this scenario reduces the number of attack operations by 12-28% across versions, but reduces the number of adversaries that could launch such attack operations drastically (over 65% for the OEM devices). PolyScope is available as open source on Github[2]. We have reported all vulnerabilities.

This paper makes the following contributions:

2. PolyScope Repository: http://github.com/yxl74/PolyScope

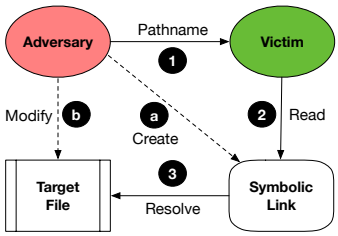

Fig. 1. **ContactsProvider Vulnerability**: (1) Adversary provides pathname to victim (as URI) to (2) lure the victim to an adversary-created symbolic link (a) that (3) the victim resolves to the target file enabling the adversary to modify the file indirectly through the victim (b).

- *We present the* PolyScope *analysis tool to triage Android filesystem access control policies, accounting for permission expansion and specialized controls in Android systems.* PolyScope reduces Android's access control policies to the set of attack operations that adversaries are authorized to launch.
- *We extend* PolyScope *to compute attack operations accounting for the newly added Scoped Storage defense.* We show how PolyScope can be extended to reason about Scoped Storage and other *restrictive* access control policies independently.
- *We use* PolyScope *to triage eight Google and OEM Android releases to assess the effectiveness of Scoped Storage quantitatively.* We find that Scoped Storage reduces the number of attack operations significantly (54-71%), but Scoped Storage is not fully employed, preventing more reduction (12-28%) and exposing two vulnerabilities.

2 Motivation

In this section, we motivate the goals of our work by using an example to show the conditions when filesystem vulnerabilities may occur and then we outline our goal of assessing the impact of Scoped Storage to reduce attacks in external storage on Android devices.

2.1 An Example Vulnerability

A recent vulnerability discovered in Android services using the ContactsProvider allows untrusted apps to gain access to privileged files [14]. The ContactsProvider 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 ContactsProvider 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 ContactsProvider) (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, modify, and delete the target file (b) to which the adversary normally lacks access.

This vulnerability occurred because adversaries of the service running ContactsProvider have the permission to

create a symbolic link in a directory to which the service also has access. The file squatting attack found by Checkpoint [30] that is described in the Introduction is caused by the same conditions, although in this case the adversary creates a file instead of a symbolic link in step (a).

2.2 The Android External Storage Problem

Filesystem vulnerablities have been a particular problem in the external storage partitions[3] of Android systems. The Android external storage partition provides a filesystem for apps to store application-specific data.

As users may benefit from the ability of multiple apps to access the same data (e.g., to edit photos and other media generated by other apps), Android provides the ability for multiple apps to share access to external storage. However, researchers have shown that vulnerabilities are caused when multiple mutually untrusting subjects can modify the same directory [49], and many vulnerabilities have been found in external storage, such as the Checkpoint vulnerability [30].

To address these concerns, Android 10 introduced an experimental defense mechanism called Scoped Storage to restrict app access to external storage [26]. Scoped Storage classifies external storage into package-specific private and shared directories, each with restricted access rights. Private directories can only be accessed exclusively by the associated app. By default, owners of files and directories in shared directories have full read and write access. However, obtaining write access to objects in shared directories owned by other apps requires either a specific Android permission limited to approved packages or explicit user consent. For a more detailed explanation of how the Scoped Storage defense works, please refer to Section 6.1.

In this paper, we address the question of how to extend access control policy analysis to determine the effectiveness of Scoped Storage in mitigating filesystem vulnerabilities.

An additional challenge arises due to the fact that not all apps will adopt the Scoped Storage defense. Some apps may still operate using earlier defenses, now referred to as *legacy storage*. Apps utilizing legacy storage have the ability to write to files belonging to other apps and may allow other apps to write to their own files, potentially introducing vulnerabilities. Therefore, we also evaluate the impact of legacy storage on the security of the external storage.

3 Background

We describe access control policy analysis methods and describe limitations in current analysis techniques.

3.1 Access Control Policy Analysis

Access control policy analysis [22, 38] involves computing authorized information flows between subjects and objects based on a system's access control policies. An access control policy authorizes an *information flow from a subject to an object* if the subject is allowed to perform a write-like operation that modifies the object, and it authorizes an *information flow from an object to a subject* if the subject is allowed to perform a read-like operation that uses the object's data (e.g., read or execute). Certain operations can be both read-like and write-like, enabling bidirectional information flow.

3. The external storage partition originally reflected a separate storage device (e.g., SD card), but modern Android systems now host the external storage partition on device storage.

However, modern Android systems typically have a large number of access control rules, resulting in a multitude of authorized information flows. To address this, researchers have developed access control analyses to identify secrecy [7, 13, 50, 51, 1] and integrity problems [24, 8]. The vulnerability discussed in Section 2.1 exemplifies an integrity problem, where an adversary controls a filesystem resource used by a victim to carry out the attack.

To detect integrity problems, access control analyses draw inspiration from integrity models, such as Biba integrity [2], to identify information flows from adversary processes to victim processes. These information flows, referred to as *integrity violations* (IVs), are formally defined as tuples consisting of the resource, adversary, and victim. An IV occurs when the access control policy permits 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 an information flow from the resource to the victim (i.e., the victim is authorized to perform a read-like operation on the resource).

3.2 Limitations of Current Techniques

Access control policy analyses attempt to solve three main problems to help identify vulnerabilities, but current approaches suffer from key limitations on each problem.

The first problem is to **characterize subjects and objects properly given multiple access control policies**. Researchers have recently proposed techniques to reason about MAC and DAC policies in combination [7, 19], but they have not considered how to compose subjects and objects from multiple policies systematically. Making matters worse, the new access control policy added by Scoped Storage defense differs significantly from existing Android MAC and DAC policies, further complicating the accurate characterization of subjects and objects across multiple policy models.

The second problem in using access control policy analysis is to **identify the permissions that adversaries could control to launch attacks comprehensively**. A problem is that adversaries may exploit the flexibility in policy models like DAC and Android permissions to add permissions that create more integrity violations. Adversaries may either obtain additional Android permissions from unsuspecting users or may grant permissions to objects they "own" to potential victims to lure them into attacks. Researchers have previously identified problems caused by DAC policy flexibility [18, 29] that limit its ability to prevent unauthorized access. While in theory MAC policies could be configured to prevent changes in DAC policies from allowing new attack operations, MAC policies often allow such changes to avoid denying desired functionality.

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 prevent attack operations in some cases, such as by prohibiting the use of symbolic links in external storage.

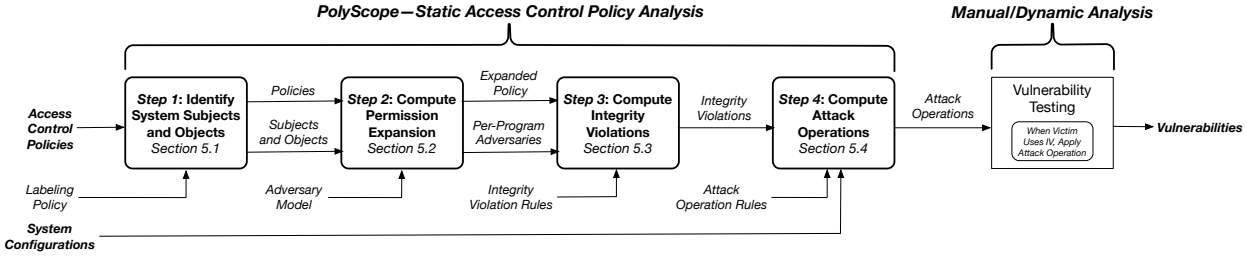

Fig. 2. POLYSCOPE **Logical Flow:** POLYSCOPE construct subjects/objects based on labeling policy (Step 1), permission expansion by those adversaries (Step 2), the integrity violations to which adversaries are authorized (Step 3), and the attack operations adversaries may perform to launch attacks (Step 4) as test cases for vulnerability testing.

4 POLYSCOPE OVERVIEW

In this paper, we present an 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. POLYSCOPE triages the *access control policies* and *system configurations* for the particular system under test and produces a set of *attack operations* that should be vetted in vulnerability testing.

Figure 2 shows POLYSCOPE's approach, where the two user (analyst) inputs, Android access control policies and Android system configurations, are highlighted in bold[4]. The sources of other inputs are described below.

In Step 1, POLYSCOPE maps processes and filesystem resources to unique subjects and objects for the access control policies using a *labeling policy*. Since Android access control combines policies in a restrictive manner (i.e., all policies must authorize an operation), labeling can be done independently for each policy model, which makes it straightforward to extend Android access control with Scoped Storage as described in Section 6.1. In Step 2, POLYSCOPE determines the permissions that may be associated with subject and object labels by modeling how each subject's adversaries may expand the permissions available to themselves and their victims by exploiting the flexibility in Android and DAC access control policies, as described in Section 6.2. Adversary models are also chosen once for the combination of policies, where we show the adversary model we use for Android systems in the Threat Model in Section 5. In Step 3, POLYSCOPE uses these expanded permissions to compute integrity violations based on *integrity violation rules* defined in Section 6.3. In Step 4, POLYSCOPE uses these integrity violations to compute the attack operations possible using *attack operation rules* that reference additional *system configurations* as defined in Section 6.4. These rule sets are defined by POLYSCOPE and are independent of the policy model.

Attack operations computed in Step 4 identify all the operations that adversaries are capable of performing to modify resources to launch attacks. Using the computed attack operations, an analyst can perform vulnerability testing on victim applications either manually or preferably using dynamic analysis. In Section 7, we describe a basic dynamic analysis analysis method to detect victim use of resources that can be modified by attack operations, from which we find two new vulnerabilities from subsequent manual testing in Section 8.5.

4. The specific inputs are described in "Data Collection" in Section 7.

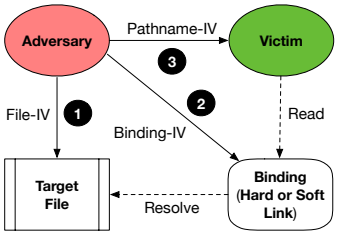

Fig. 3. **Integrity Violation (IV) Classes**: (1) *File-IVs* grant adversaries direct access to modify files that victims use; (2) *Binding-IVs* grant adversaries the ability to modify name resolution of file names; and (3) *Pathname-IVs* enable adversaries to lure victims to the part of the filesystem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

5 THREAT MODEL

In this paper, we assume that adversaries may modify any part of the filesystem to which they are authorized by the combination of Android access control policies, which POLYSCOPE computes as *integrity violations*. In addition, we assume that adversaries will perform any operation to launch attacks on integrity violations that are possible given the system configuration, called *attack operations*. In this section, we examine the specific types of integrity violations and attack operations we consider as threats in this paper.

Based on the example in Figure 1, we show the three classes of integrity violations (IVs) we consider in developing POLYSCOPE in Figure 3 on filesystem access, covering a wide variety of vulnerabilities including confused deputy [17] and time-of-check-to-time-of-use (TOCTTOU) vulnerabilities [31, 3]. 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. File-IVs may be distinguished further 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. 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

TABLE 1
Google's Process Privilege Levels [15]

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 8.

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 that a victim may use. Pathname-IV attack operations lure a victim who processes adversary-controlled pathnames (e.g., URLs via IPCs) to an adversary-controlled binding to exploit a link traversal, which we call a *luring 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. For example, we trust the Android operating system to satisfy the reference monitor concept [21]. Second, our assumptions about trust among user-space processes is determined by Google's Process Privilege Levels shown in Table 1. A subject trusts services/apps at its privilege level or higher. Other subjects are adversarial. We have shown how to validate that the Android policy is consistent with the trust implied by these privilege levels [27].

6 POLYSCOPE DESIGN

In this section, we examine the design challenges in computing attack operations for Android systems. In particular, we focus on four key steps outlined in the POLYSCOPE overview in Section 4.

6.1 Identify Subjects and Objects

The first step is to identify the subjects and objects in the system based on the combination of access control policies. Access control determines when a process may perform an operation on a system resource. However, access control policies are typically expressed in terms of identifiers for subjects and objects (e.g., user IDs and group IDs), rather than individual processes and resources, to enable access control decisions to be made as the system's processes and resources evolve dynamically. The mapping of access control identifiers to system processes and resources forms a *labeling policy*. Since Android uses a combination of access control policies, a question is how to determine the combination of identifiers (i.e., the labeling policy) that constitute subjects and objects for this combination of policies.

Researchers have previously found it useful to identify combinations of policies that are either restrictive or authoritative [52]. A composition of policies is said to be *restrictive* when any policy can deny access (i.e., the authorized permissions are the *intersection* of each policy's permissions). On the other hand, a combination of policies is said to be *authoritative* if any policy may grant a permission, even if it is denied by another policy (i.e., the authorized permissions are the *union* of each's authorized permissions). In either case, the subjects and objects for each policy can be determined independently and the resultant permissions are a simple composition (e.g., intersection or union) of the permissions assigned to the individual policies (i.e., between each policies' subjects and objects).

We find that the combination of Android access control policies is restrictive. Android requires that all policies must authorize an operation for it to be permitted. As a result, we can simply extend the subjects and objects in POLYSCOPE by specifying the Scoped Storage subjects and objects identifiers independently from the other Android policies.

The subject and object identifiers used by POLYSCOPE for the Android access control policies, other than Scoped Storage, are as follows[5].

- **SELinux Type Enforcement (TE)**: Subjects and objects are assigned TE labels.
- **SELinux Multilevel Security (MLS)**: Subjects and objects are assigned MLS category sets.
- **UNIX Discretionary Access Control (DAC)**: Subjects are assigned DAC User ID and a set of DAC groups (i.e., an owner Group ID and supplemental group IDs). Objects are assigned an DAC UID and DAC GID for its owner.

The Scoped Storage defense (see Section 2.2) has been designed to control access to each app's files in the Android external storage partition. Recall that Scoped Storage separates app-specific storage into private and shared directories. Private directories may only ever be accessed by the owning app, but other apps may gain access to shared directories through Android permissions or user consent. The apps may obtain read (only) access to any shared directories through the `READ_EXTERNAL_STORAGE` (REX) Android permission. Write access may be obtained in two ways. First, an app may gain write access to files in a shared directory by obtaining user consent through the Android APIs including MediaStore, Storage Access Framework (SAF) and Photo Picker. Second, apps that are vetted prior to publication in the Google Play Store are eligible to obtain the `MANAGE_EXTERNAL_STORAGE` (MES) permission that grants them read and write access to all resources in shared and legacy directories, but not private app directories.

For compatibility purposes, some apps may be declared as *legacy* apps, which basically means that these apps use the access controls that predate Scoped Storage. Legacy apps may gain read and write access to any package's shared directories using the `WRITE_EXTERNAL_STORAGE` (WEX, deprecated since Android 11) permission. In ad-

5. For details justifying these choices, see the original POLYSCOPE paper [27].

dition, legacy apps are allowed to place files in the root directory of external storage, which is shared among legacy apps and other system services. This root directory is not accessible to apps compliant with Scoped Storage without the MES permission.

As a result, the Scoped Storage access control policy governs access for each app based on its identity (i.e. package name), its Android Permissions, and whether it complies with Scoped Storage. Specifically, the subjects and objects of the Scoped Storage policy are defined below.

- **Scoped Storage Subject:** Each subject is defined as a combination of: (1) an owner identity (i.e., app package name); (2) whether the app opts for Scoped Storage (i.e., is a legacy or Scoped Storage subject); and (3) the app's Android permissions (i.e., REX and MES for Scoped Storage subjects and REX and WEX for legacy subjects).
- **Scoped Storage Object:** Each object is defined by: (1) its owner (i.e., package name) and (2) its containing directory type (i.e., private or shared or legacy).

Given the Scoped Storage subject and object definitions, we summarize the Scoped Storage access rules in Table 2. Only the owner of a private directory can access those objects, except through IPC sharing[6]. Scoped Storage subjects can access objects they own in the shared directory by default, but need the REX permission to read objects owned by other subjects. Scoped Storage subjects with the MES permission can access all objects in the shared and legacy directories. Legacy (non-Scoped) subjects can read and write all objects in the shared and legacy directories with the WEX permission [7].

Because access control policies in Android are restrictive, we can add the subject and object definitions for Scoped Storage independently from other policies. In analysis, POLYSCOPE only needs to intersect the outcome of the authorization using the Scoped Storage policy with that of the MAC (TE and MLS) and DAC authorization results to determine which subjects can access an object.

6.2 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, which we call *adversary permission expansion*, and (2) adversaries may delegate DAC permissions for objects that they own to potential victims, which we call *victim permission expansion*. We describe how these two forms of permission expansion manifest in Android and examine the expansion allowed for Scoped Storage and legacy adversaries.

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, 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 often allow apps to gain privileges associated with Android permissions, 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. Scoped Storage adds another kind of adversary permission expansion by allowing apps to declare themselves as legacy apps. The legacy flag grants write privilege to files in multiple locations of external storage and greatly boosts an adversary's capability to launch attacks. One of the vulnerability case studies we highlight in Section 8.5 exploits the use of the legacy flag for 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 in a typical DAC protection system is undecidable in the general case [18]. As a result, researchers explored alternative DAC models within which the safety problem could be solved, such as the take-grant model [28], the typed access matrix [36], and policy constraints [42]. However, adversaries may still grant victims their permissions

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, when adversaries own these directories, they can simply grant the removed permissions to potential victims.

Scoped Storage Impact on Permission Expansion: Scoped Storage permits two kinds of adversary permission expansion. First, apps that can obtain the MES permission can modify any file in a shared or legacy directory of external storage. Fortunately, Google must vet any app before it can even request that permission, but even some vetted apps only have a T1 Google privilege level (see Table 1), exposing some risks. Thus, we assume any subjects requesting the MES permission must have been vetted for that permission.

Second, apps can request write access to files from users (i.e., request user consent). While users may grant access to any file in a shared directory, in general, the impact of Scoped Storage is largely bypassed for resources in shared directories if that is done comprehensively. In this work, we do not apply user consent of access to individual files in shared directories to permission expansion. Studying possible risks of such user consent is future work.

For pre-Scoped Storage systems, we assume that victims can expand permissions (i.e., perform adversary expansion) to obtain the REX/WEX permissions since most apps

6. Sharing through Android ContentProvider requires programs to actively share their files (i.e., passing file descriptors to opened files), so we leave the potential threat of active sharing as future work.

7. Granting an app the WEX permission also grants the REX permission for reading. All legacy apps we have reviewed have requested the WEX permission.

TABLE 2
Subject to Object Access in Scoped Storage

Subject Type[1]	Private Objects	Shared Objects	Legacy Objects
Owner Scoped Subjects	R/W	R/W	No Access
Other Scoped Subjects	No Access	R with REX	No Access
Other Scoped Subjects with MES	No Access	R/W	R/W
Other Legacy Subjects	No Access	R with REX, RW with WEX	R with REX, RW with WEX

[1] Owner Scoped Subject is a subject whose package name matches the package name the objects. Other Scoped/Legacy Subjects refer to any subject whose package name does not match that of the objects.

need access to shared folders in external storage. However, for post-Scoped storage systems, victims no longer need to declare REX/WEX to access their own files stored in external storage. Adversaries cannot cause any form of victim permission expansion because they cannot authorize REX/WEX to the victim programs. Outside of external storage, both types of permission expansion threats still remain.

6.3 Compute Integrity Violations

We show how to compute integrity violations for file-IVs, binding-IVs, and pathname-IVs defined in Section 5.

Computing File Integrity Violations: A file integrity violation occurs when a victim subject has permission to use (i.e., read, write, or execute) a file object that an adversary subject also has permission to modify. In practice, many subjects read file objects that their adversaries may write (`read-IVs`). However, the risks increase when a subject executes (`exec-IVs`) or modifies such files (`write-IVs`). For exec-IVs, executing input from an adversary allows the adversary to control the victim's executable code. In the case of write-IVs, if a subject writes to a file object that its adversaries can also write to, the adversaries may be able to undo or replace valid content.

This rule determines whether the victim is authorized by the combination of access control policies for reading, writing, or executing file objects, 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 object (`write` predicate), then the associated integrity violation is created.

Computing Binding Integrity Violations: A binding integrity violation occurs when a subject may use a binding object that adversaries can modify in resolving a file pathname.

This rule parallels the rule for file-IVs, except that this rule applies to a victim having the permission to use a binding object 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.

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 object meets the requirements of a binding-IV (see above), then a pathname-IV is also possible for this victim.

6.4 Compute Attack Operations

We define how POLYSCOPE computes attack operations from the integrity violations produced 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.

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.

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.

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

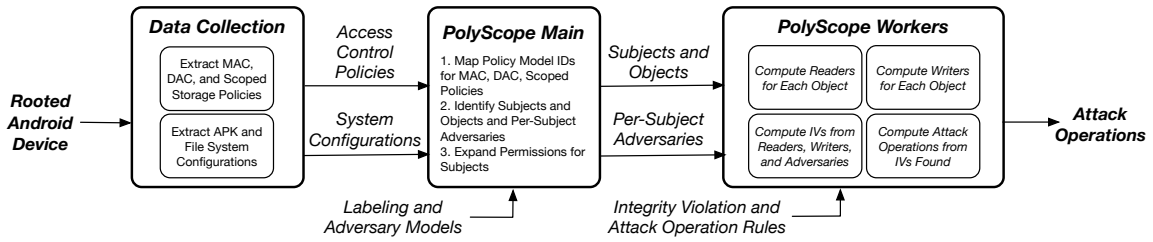

Fig. 4. POLYSCOPE **Implementation:** POLYSCOPE collects inputs from rooted Android devices to identify subjects and objects. IVs and attack operations are computed in parallel per object by POLYSCOPE workers.

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.

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.

7 IMPLEMENTATION

The POLYSCOPE tool is implemented fully in Python in about 3300 SLOC and is compatible with Android version 5.0 and above. The POLYSCOPE implementation is shown in Figure 4. First, POLYSCOPE extracts access control policies and system configurations automatically in a Data Collection phase. Then, POLYSCOPE's Main phase identifies all the subjects and objects as described in Section 6.1, determines the per-subject adversaries according to the Google Process Privilege Levels in Table 1, and expands subject permissions as described in Section 6.2. Next, POLYSCOPE Workers computes IVs as described in Sections 3.1 and 6.4, respectively. We are able to parallelize these steps per object which has a significant performance impact, as described in Section 8.6. Additional implementation details are the same as for the original POLYSCOPE tool [27].

Data Collection: POLYSCOPE has a variety of data collection scripts to collect access control polices (i.e., MAC, DAC, Android permissions, and Scoped Storage) and system configurations (i.e., filesystem settings and FileProvider use) to provide inputs to POLYSCOPE. The methods are relatively straightforward for accessible files and processes, as described previously [27]. However, POLYSCOPE scripts 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. [19] is able to extract MAC policy and part of DAC configuration from Android firmware images without rooting devices. However, this approach cannot extract all files located in some directories like `/data`. As shown in Table 1 of their paper [19], about 75% of the files' DAC configuration in `/data` cannot be retrieved, which we extract with our scripts.

Data collection for Scoped Storage requires collecting access control information for each package and Scoped Storage directory. To collect packages and their Android permissions, POLYSCOPE queries the PackageManager service for all the APKs on the device. Then, POLYSCOPE parses the Android manifest files for the extracted APKs to obtain the permission mapping. For Scoped Storage directories, POLYSCOPE extracts the Scoped Storage database file owned by MediaProvider to retrieve the owner of each external storage resource by package name. 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.

POLYSCOPE **Main:** This POLYSCOPE component controls the steps in the POLYSCOPE analysis. As shown in Figure 4, the POLYSCOPE Main component runs three computations in series to identify subjects and objects and expand permissions, as described in Sections 6.1 and 6.2, respectively. The most difficult step is to map the subjects between MAC (labels), DAC (UIDs/GIDs), and Scoped Storage (packages) policies. To find the mapping between UIDs and packages, POLYSCOPE parses the `package.list` file. However, we found that some package name-to-UID mappings are not one-to-one, as we expected and saw for MAC-to-DAC mappings, as multiple package names can be mapped to the same UID. In this case, POLYSCOPE over-approximates the mapping by assigning the union of all the package names that map to this UID for the subject.

POLYSCOPE **Workers:** Using the set of subjects, their adversaries, and objects, POLYSCOPE can now compute the attack operations. The POLYSCOPE implementation decomposes this computation into discrete components that can be parallelized, as shown in Figure 4. First, workers compute the subjects that can read (i.e., read and execute) and write each object. This computation run per policy model (DAC, MAC, Scoped Storage) and the results per object are intersected. Given the readers and writers for each object and policy model, the IVs for each object can be computed in parallel, one object per worker to roughly balance the load per worker. Finally, the attack operations that applyfor each object's IVs can be computed in parallel as well. We validated the attack operations found can be performed 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. 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 [20]. 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 8.5 manually. We discuss the challenges in automating vulnerability testing in Section 9.

8 Evaluation

In this section, we focus on measuring the impact of the Scoped Storage defense on threats to Android systems. Note that we evaluated PolyScope against prior systems [19] in our previous paper [27]. Here, we apply the updated PolyScope tool described in this paper to six fresh installs of Android releases that employ Scoped Storage[8] (version 11.0 and above) and two fresh installs of Android releases that do not employ Scoped Storage (version 9.0).

We explore the following research questions:

- **RQ1**: What fraction of the total number of threats in Android systems occur in external storage before and after the addition of Scoped Storage?
- **RQ2**: How does Scoped Storage impact the types of integrity violations and attack operations that may be attempted in external storage?
- **RQ3**: How many of the threats in external storage are due to legacy apps?
- **RQ4**: How many attack operations and attackers could be removed if all legacy apps are converted to Scoped Storage apps?

We first examine the distributions of IVs and attack operations within systems at-large and for external storage only (RQ1-RQ2). These analyses show that Scoped Storage has reduced the number of attack operations in external storage, particularly by removing victim expansion and by removing squatting attacks. However, we find that many victims remain threatened because of the use of legacy apps (RQ3). To assess the ideal impact of Scoped Storage, we evaluate the hypothetical case where all apps are compliant with Scoped Storage (RQ4). We also examine two vulnerabilities found in external storage using PolyScope and the performance of access control analysis using PolyScope in Sections 8.5 and 8.6, respectively.

8.1 Effects of Scoped Storage

RQ1: *What fraction of the total number of threats in Android systems occur in external storage before and after the addition of Scoped Storage?* Table 3 displays the integrity violation (IVs) and attack operation (Attack Ops) counts for the eight Android systems, where two systems (Pixel3a 9.0 and Galaxy S20 9.0) do not use Scoped Storage. Table 3 compares the total counts to the counts in external storage only. Note that the IV counts (**IVs**) are a sum of the number of objects that may be used to attack each victim, as described in Table 3.

The first two rows in Table 3 show the IV and attack operation counts for the whole system, showing that the number of attack operations tends to be slightly greater than the number of IVs for the system at large, although slightly lower for OnePlus systems. The next two rows in Table 3 show the IV and attack operation counts for external storage alone. For external storage, the number of attack operations is always less than the number of IVs, due to the lack symbolic links in external storage, which predates Scoped Storage, and the reduction in squatting attacks, which we show using PolyScope in Section 8.2.

Examining Table 3 from left to right, we see that the IV and attack operation counts of the pre-Scoped systems (i.e., Pixel3a 9.0 and Galaxy S20 9.0) were much higher than their respective counterparts (i.e., other Google devices and OEM devices, respectively). PolyScope shows that this reduction is largely because Scoped Storage eliminates victim expansion (see Section 6.2), as described in Section 8.2.

However, we see the reduction in the fraction of IVs and attack operations differs between Google and OEM devices. For OEM devices, while the reduction in the number of IVs and attack operations is significant (i.e., from pre-Scoped Storage Galaxy S20 9.0 to Samsung and Oneplus versions 11.0 and 12.0), these counts remain much greater than for Google devices. We use PolyScope to assess how the greater use of legacy apps (i.e., apps not compliant with Scoped Storage) in these OEM devices increases these counts (see Section 8.3) and the impact if all apps would be compliant with Scoped Storage (see Section 8.4).

8.2 Reasons Scoped Storage Reduces Threats

RQ2: *How does Scoped Storage impact the types of integrity violations and attack operations that may be attempted in external storage?* In this section, we show how PolyScope enables us to explain the reasons for any reductions in threats due to Scoped Storage from Table 3. Rows 1-2 in Table 4 show the total IV counts for the eight Android systems in Table 3 broken down for file objects (read-IVs and write-IVs) and directories (pathname-IVs and binding-IVs). Note that the write-IVs are a subset of the read-IVs and the binding-IVs are a subset of the pathname-IVs[9] Rows 3-4 in Table 4 show the same information, but for external storage only.

We can see in Table 4 that the number of IVs for both files and directories are significantly reduced when compared to the pre-Scoped Storage systems (i.e., Pixel3a 9.0 and Galaxy S20 9.0). This shows that access control decisions, such as the deprecation of the WEX permission and more limited use of the REX permission reduce threats. However, an even more obvious impact is shown in row 4, where the IV counts are the same for pathname-IVs and binding-IVs. This is caused because adversaries cannot change the permissions of other apps, i.e., victim expansion as described in Section 6.2 is no longer possible in Scoped Storage systems.

Table 5 shows the counts for the modification attack and squat attack operations described in Section 6.4 in total and for external storage only[10]. Once again the counts for attack operations in external storage is significantly lower in Scoped Storage systems than pre-Scoped Storage systems

8. Oneplus8T is a relatively new phone that does not have any pre-Scoped Storage firmware available

9. The definition of binding-IVs implies that they are a subset of the pathname-IVs, but the write-IVs happen to be a subset of the read-IVs because victims always have read permission when they have write permission in the systems we examined.

10. We do not show link traversal and luring traversal attack operations, which require symbolic links that have been banished from external storage since prior to Scoped Storage.

TABLE 3
Summary of Integrity Violations (IVs) and Attack Operations Total and in External Storage across Vendor Releases

	Google Devices			OEM Devices				
	Pixel3a 9.0	Pixel3a 11.0	Pixel3a 12.0	*Galaxy S20 9.0*	Galaxy S20 11.0	Galaxy S20 12.0	Oneplus8T 11.0	Oneplus8T 12.0
Total IVs	2,124	1,334	1,480	31,489	12,713	6,808	11,987	14,704
Total Attack Ops	2,512	1,628	1,794	36,258	15,414	8,465	11,540	14,135
Ext IVs	1,021 (48%)	260 (19%)	374 (25%)	12,679 (40%)	3,713 (29%)	2,288 (34%)	4,365 (36%)	5,532 (38%)
Ext Attack Ops	527 (21%)	241 (14%)	219 (12%)	11,336 (31%)	3,219 (21%)	1,906 (23%)	3,929 (34%)	4,454 (32%)

IVs $= \sum_{v}^{V} |IV_{obj}(v)|$, where $IV_{obj}(v)$ returns the set of objects in the IVs for a victim $v \in V$ as computed per Section 6.3.
Attack Ops $= \sum_{iv}^{\mathbf{IVs}} |OP(iv)|$, where $OP(iv)$ returns the set of attack operations for an integrity violation $iv \in$ **IVs** computed per Section 6.4.
Ext IVs (Attack Ops) are IVs (Attack Ops) whose objects are located in an external storage partition.

TABLE 4
Integrity Violations (IVs) by IV Type in Total and in External Storage across Vendor Releases

	Google Devices			OEM Devices				
	Pixel3a 9.0	Pixel3a 11.0	Pixel3a 12.0	*Galaxy S20 9.0*	Galaxy S20 11.0	Galaxy S20 12.0	Oneplus8T 11.0	Oneplus8T 12.0
Total File IVs (Read/Write)	750/149	632/164	713/154	13,248/7,213	6,281/4,186	3,036/1,797	5,865/3,415	7,248/4,681
Total Dir IVs (Pathname/Binding)	1,674/314	702/195	767/389	18,241/6,713	6,432/4,768	3,772/2,667	6,122/2,196	7,420/2,687
Ext File IVs (Read/Write)	308/149	202/132	187/111	6,569/4,384	2,758/2,080	1,503/1,049	2,944/977	3,879/1,293
Ext Dir IVs (Pathname/Binding)	713/219	58/58	187/187	6,110/4,767	955/955	785/785	1,421/1,421	1,653/1,653

IVs $= \sum_{v}^{V} |IV_{obj}(v)|$, where $IV_{obj}(v)$ returns the set of objects in the IVs for a victim $v \in V$ as computed per Section 6.3.
Ext IVs are IVs whose objects are located in an external storage partition

TABLE 5
Attack Operations by Type in Total and in External Storage across Vendor Releases

	Google Devices			OEM Devices				
	Pixel3a 9.0	Pixel3a 11.0	Pixel3a 12.0	*Galaxy S20 9.0*	Galaxy S20 11.0	Galaxy S20 12.0	Oneplus8T 11.0	Oneplus8T 12.0
Modification Attacks	750	632	713	13,248	6,281	3,036	5,865	7,284
Squat Attacks	314	195	389	6,713	4,768	2,667	2,196	2,687
Ext Modification Attacks	308	202	187	6,569	2,758	1,503	2,944	3,879
Ext Squat Attacks	219	39	32	4,767	461	403	985	575
Squat Attacks Prevented	0	19	155	0	454	382	436	1,078

Attack Ops $= \sum_{iv}^{\mathbf{IVs}} |OP(iv)|$, where $OP(iv)$ returns the set of attack operations for an integrity violation $iv \in$ **IVs** computed per Section 6.4.
Ext Attack Ops are Attack Ops whose objects are located in an external storage partition.

(Pixel3a 9.0 and Galaxy S20 9.0). Modification attacks are reduced because the number of file IVs has been reduced as discussed above. In addition, the number of squat attacks have been reduced because Scoped Storage prevents victims from accessing adversary-created files by default (i.e., without REX or MES permission). In row 5 of Table 5, we show the count of the number of binding-IVs that cannot be converted into squat attacks because the victims lack REX or MES permissions to access adversary-controlled directories. As a result, recent vulnerabilities, such as the Man-in-the-Disk [30] that leverage squat attack operations, are no longer possible by default in Scoped Storage releases.

8.3 Problems with Legacy Applications

RQ3: *How many of the threats in external storage are due to legacy apps?* Table 6 shows a comparison of IVs (i.e., broken down further into victim subject and object counts) created by apps that are compliant with Scoped Storage and those that are not, which are called *legacy apps*. Here, we see that a modest number of pre-installed legacy apps across vendors (row 1) causes over twice the number of subjects to become potential victims (i.e., have at least one IV due to a legacy app) of attacks (row 2) due to nearly twice the number of object (row 3) than for compliant apps (rows 4-6). This is not surprising since the access permissions of legacy apps is similar to Scoped Storage apps with MES Android permission. But, when applying Scoped Storage, the MES Android permission is only granted to applications that have been vetted by Google, which limits the number of third-party apps that may obtain that permission and presumably improves the trust in such apps.

8.4 Fully-Enforced Scoped Storage

RQ4: *How many attack operations and adversaries could be removed if all legacy apps are converted to Scoped Storage apps?* To measure how well Scoped Storage could potentially work to reduce attack operations, we move objects in legacy locations into the shared folders protected by Scoped Storage (e.g., ownership info tracked by MediaProvider). Since the *WRITE_EXTERNAL_STORAGE* permission has been deprecated beginning with Android 11, only file owners have write access to the files in shared folders. Then, we assume legacy flags are removed and perform PolyScope analysis to compute attack operations on external storage only. The analysis results are shown in Table 7, where the row 1 is the original attack operation count for external storage and row 2 is the attack operation count for external storage after the procedures described above, and rows 3 and row 4 count the corresponding changes in the number of adversaries.

We see that the number of attack operations decreases 12%-28%, but the number of adversaries decreases more significantly: at least 36% for Google devices and at least 65% for all other OEM devices. This suggests that the level of decrease in attack operations does not reflect the corresponding reduction in the number of adversaries. We observe that the remaining adversaries are file management apps given MES and REX permissions in Scoped Storage, which conflicts with their low privilege classification under the Google Privilege Levels [15] in Table 1. It is future work to assess whether permissions should be refined further to reduce attack operations or additional privilege levels need to be added to accommodate such apps.

8.5 Vulnerability Case Studies

Using the attack operations computed by PolyScope, we manually identified two previously unknown vulnerabilities that we describe below as well as other resources that face significant risks. We have ethically reported these vulnerabilities.

Replace Over-the-Air Updates: We found a new vulnerability in the Oneplus 8T system running Android 11 release. We found that Oneplus temporarily stores an OTA update

file in a hidden folder located in the root directory of external storage (i.e., accessible to legacy apps). We found that untrusted applications that request the legacy apps can observe the OTA download and replace the OTA update file before installation takes place. This could potentially grant root privilege to attackers with a properly engineered OTA update file. POLYSCOPE further identified many other objects stored in legacy location vulnerable to adversarial legacy apps. These files include configuration files, log files, cache files, and cookies. We did not fully explore how these attack operations can be exploited, but it is extremely dangerous for privileged apps to store their data files in locations accessible to legacy apps. The above vulnerability shows the danger of legacy apps, and how OEMs may not use external storage correctly in the face of legacy apps.

Malicious Code Execution: We found potential vulnerability related to the Quick App feature that is widely used by major Chinese OEMs including Huawei, Lenovo, Oneplus and Xiaomi. Quick App is a lightweight framework that allows users to access simple services (i.e., weather, taxi, payment) without installing heavyweight APKs. One of the most used features is gaming, where users can start playing with one simple click. On the Oneplus 8T Android 11 device we tested, POLYSCOPE found that the Quick App framework stores executable files in a hidden folder located in a legacy location, where malicious legacy apps can easily squat. The Quick App framework is a highly privileged victim running as a pre-installed platform app. We did not fully evaluate how much damage we can done to the system, but we are able to cause the Quick App service to restart by corrupting the game files.

8.6 POLYSCOPE Analysis Performance

We measured the performance of POLYSCOPE in analyzing the six Android releases supporting Scoped Storage. The overhead was measured on a Mac M1 Pro (10 cores) with 16GB of RAM. We measure the performance of the POLYSCOPE Main and Workers to compute attack operations as described in Section 7. Unlike the original POLYSCOPE tool [27], we use a multi-process implementation to leverage parallelism for POLYSCOPE Workers, as described in Section 7. Initially, we divided the objects among the workers evenly, but we found that this results in an unbalanced load as some objects have many more readers and writers than others. As a result, we assign the objects one at a time to workers, which tends to balance the load. Figure 5 shows the performance results. We measure the performance of POLYSCOPE's analysis using 1 to 64 processes. We can see that performance benefits significantly as we increase the number of processes, from over 6,000s for a single process to a maximum of 524s for 64 processes for the six Android systems.

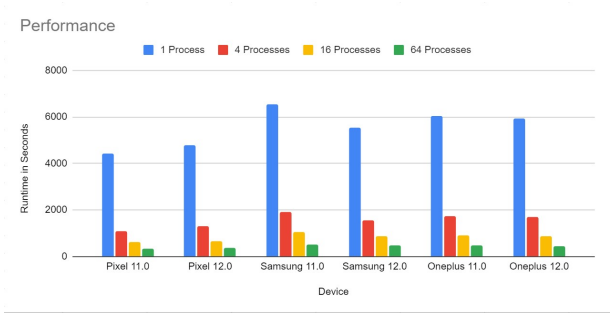

Fig. 5. POLYSCOPE Analysis Performance

9 DISCUSSION

Limitations of POLYSCOPE: We identify three limitations of the current POLYSCOPE tool: (1) POLYSCOPE requires a rooted phone to collect filesystem data; (2) POLYSCOPE cannot always determine the mapping among multiple access control policies for all subjects; (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. [19] proposed BigMAC, which includes a technique to extract accurate DAC configuration data from these privileged directories (∼95%). However, we found that BigMAC cannot extract files effectively from some directories, such as `/data`, as described under Data Collection in Section 7. We will explore methods to achieve complete recovery in future work.

Another limitation of POLYSCOPE is that we cannot map all processes to complete subjects as defined in Section 6.1. The main problem is to connect the package name and UID from the package list (see Section 7 for POLYSCOPE Main) to the MAC label/MLS category. Recall that the mapping between UID and MAC label is determined by running the program [7], but not all packages listed can be run, as some have abstract names. We compare the number of entries in the package list to the number of unique subjects we compute per system. For Google devices, the count is the same, but for OEM devices up to 10% of the package list entries are unmapped (for Samsung Galaxy 11.0 and 12.0). We will explore how to complete this mapping in the future.

Finally, POLYSCOPE lacks a systematic way to test the victims for vulnerabilities to the attack operations found. There are two problems to address. First, we need to know when a victim may use a resource that is associated with an attack operation. This is difficult to determine statically. The STING system [49] provides passive runtime monitoring of processes for use of bindings that could be used to perform file squatting and link traversal attacks using DAC policies, so such a runtime monitoring approach could be extended to utilize attack operations generated by POLYSCOPE. Second, once we know when a victim may be threatened by an attack operation, we need to generate test cases that could exploit the victim. Current fuzzing techniques [20] do not target these types of attack operations. Runtime monitoring techniques [49, 46] and similar techniques for assessing use of Android intents [1] generate simple test cases, enabling detection of unprotected cases. We aim to generate test cases that account for the conditional checks in the program fully.

Limitation of Scoped Storage: In terms of Scoped Storage, most of the security problems are caused by OEMs not following the safe guidance or using the legacy flag as shown in Section 8.3. We expect the problem to ease and disappear when Scoped Storage is fully enforced. However, attack operations caused by third party application with MES shown in Section 8.4 will still exist and we believe that a more fine-grained permission control is needed. Potential methods include new data access API specifically

TABLE 6
Number of Apps and Their Victims (for Attack Operations) for Scoped-Apps vs. Legacy-Apps

	Google Devices		OEM Devices			
	Pixel3a 11.0	Pixel3a 12.0	Galaxy S20 11.0	Galaxy S20 12.0	Oneplus8T 11.0	Oneplus8T 12.0
Legacy-App Count	11	18	44	44	23	22
Victims of Legacy-Apps[1]	141	254	280	286	275	293
Object Count	6	4	44	24	53	57
Scoped-App Count	106	252	230	205	222	215
Victims of Scoped-Apps[2]	69	124	120	125	113	108
Object Count	4	2	12	10	31	33

Unit: Subject Count
[1] Number of unique victims with IVs where an adversary is a legacy app.
[2] Numbers of unique victims with IVs where an adversary is a compliant app

TABLE 7
Attack Operation Comparison between Current Systems and Fully Enforced Scoped Storage

	Google Devices		OEM Devices			
	Pixel3a 11.0	Pixel3a 12.0	Galaxy S20 11.0	Galaxy S20 12.0	Oneplus8T 11.0	Oneplus8T 12.0
Ext-Storage Attack Operations	241	219	3,219	1,906	3,929	4,454
Full-Scoped Attack Operations[1]	173(-28%)	166(-24%)	2,831(-12%)	1,620(-15%)	3,222(-18%)	3,564(-20%)
Ext-Storage Adversaries	25	22	62	57	61	63
Full-Scoped Adversaries[2]	16 (-36%)	9(-59%)	18(-70%)	18(-68%)	21(-65%)	16(-74%)

[1] Fully enforced Scoped Storage attack operation count
[2] Numbers of unique attackers after Scoped Storage is fully enforced

for MES apps or new resources protection technique for all applications. An intermediate solution will be limiting MES permission usage while apps are in the background, or notify users when apps are using MES.

10 Related Work

Researchers have long been aware of the three types of integrity violations listed in Section 5, but have encountered difficulties in defending vulnerabilities associated with such violations. Various mechanisms have been proposed to prevent attacks during name resolution, including defenses against binding and pathname vulnerabilities. These defenses have often focused on TOCTTOU attacks [31, 3, 11]. Some defenses are implemented as program or library extensions [9, 34, 10, 43], while others are implemented as kernel extensions [25, 35, 6, 33, 44, 45]. The methods overlap, with some enforcing invariants on file access [9, 25, 45, 34, 35, 44], some enforcing namespace invariants [6, 33], and some aiming for "safe" access methods [10, 43]. In general, program defenses have been limited by their lack of insight into the system state, while system defenses have been limited by their lack of side-information about the program's intent [5].

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 [18], so techniques to restrict DAC [28, 36, 42] and apply mandatory access control (MAC) enforcement [12, 2] were then explored, culminating in MAC enforcement systems, such as Linux Security Modules [52] employed by SELinux [37] and AppArmor [32]. Researchers then proposed MAC enforcement for Android systems [53, 4], so a version of SELinux [37] targeting Android was developed, called Security Enhanced (SE) Android [40]. However, the attack operations we find in this paper abuse available MAC enforcement. While these techniques have been developed to limit the permissions available to individual system calls [39, 48], 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 [22, 38], but traditionally, access control policy analysis methods only reason about one policy, such as a mandatory access control (MAC) policy [38, 24, 8, 47] or an Android permission policy [13, 50, 51]. 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.* [7] were the first work that we are aware of to combine MAC and DAC policies in access control policy analysis. Hernandez et al. [19] 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.

11 Conclusions

Android uses filesystem access controls to ensure platform integrity. This paper proposes PolyScope, which enables triaging of Android systems for filesystem vulnerabilities using its combination of access control policies. Android currently employs a combination of mandatory (e.g., SE-Android) and discretionary (e.g., Unix permissions) access control policies, as well as specialized access controls like Android permissions. Android recently introduced the Scoped Storage defense to prevent vulnerabilities for external storage shared among apps. In this paper, we show how PolyScope can be extended with restrictive policies independently to easily add the ability to analyze Scoped Storage in combination with existing policies. We applied PolyScope to eight Google and OEM Android releases, finding that Scoped Storage effectively reduces Android's attack surface. However, legacy apps still pose problems, causing many attack operations and two new vulnerabilities. OEMs must expedite making privileged apps compliant with Scoped Storage.

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] Kenneth Biba. Integrity Considerations for Secure Computer Systems. Technical report MTR-3153, MITRE, April 1977.

[3] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] Shaoyong Du, Xin Liu, Guoqing Lai, and Xiangyang Luo. Watch out for race condition attacks when using android external storage. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 891–904, Los Angeles, CA, USA, 2022. URL: https://doi.org/10.1145/3548606.3560666.

[12] Bell Elliott and Leonard La Padula. 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.

[13] 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.

[14] Google. Android Security Bulletin-July 2022. July 2022.

[15] Google. Security Updates and Resources. 2022. URL: https://source.android.com/docs/security/overview/updates-resources#process_types. Accessed Dec. 10, 2022.

[16] Google. Storage Updates in Android 11. URL: https://developer.android.com/preview/privacy/storage. Accessed June 2022.

[17] 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.

[18] Michael Harrison, Walter Ruzzo, and Jeffrey Ullman. Protection in Operating Systems. *Communications of ACM*, August 1976.

[19] 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.

[20] 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.

[21] Trent Jaeger. Reference monitor. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed*, pages 1038–1040. Springer, 2011.

[22] 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.

[23] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symp.* August 2003.

[24] 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.

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

[26] Yu-Tsung Lee, Haining Chen, and Trent Jaeger. Demystifying Android's Scoped Storage Defense. *IEEE Security & Privacy*, 19(5), 2021.

[27] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Daimeng Wang, Zhiyun Qian, Giuseppe Petracca, and Trent Jaeger. PolyScope: Multi-policy access control analysis to compute authorized attack operations in Android systems. In *Proceedings of the 30th USENIX Security Symposium*, August 2021.

[28] Richard Lipton and Lawrence Snyder. A Linear Time Algorithm for Deciding Security. In *Proceedings of the* 17^{th} *Annual Symposium on Foundations of Computer Science*, 1976.

[29] Peter Loscocco *et al.* 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] OpenWall Project - Information Security Software for Open Environments. http://www.openwall.com/, 2008.

[34] Jongwoon Park, Gunhee Lee, Sangha Lee, and Dongkyoo Kim. RPS: An Extension of Reference Monitor to Prevent Race-Attacks. In *Advances in Multimedia Information Processing*, 2004.

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

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

[37] SELinux. -. URL: https://github.com/SELinuxProject. (Accessed Dec 2022).

[38] SETools. URL: https://github.com/TresysTechnology/setools. Accessed Dec 2022.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

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

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

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

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

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

[53] 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.