ScopeVerif: Analyzing the Security of Android's Scoped Storage via Differential Analysis

Zeyu Lei*, Güliz Seray Tuncay†, Beatrice Carissa Williem*, Z. Berkay Celik*, Antonio Bianchi*
*Purdue University, †Google
lei76@purdue.edu, gulizseray@google.com, beat.wil105@gmail.com, zcelik@purdue.edu, antoniob@purdue.edu

***Abstract*—Storage on Android has evolved significantly over the years, with each new Android version introducing changes aimed at enhancing usability, security, and privacy. While these updates typically help with restricting app access to storage through various mechanisms, they may occasionally introduce new complexities and vulnerabilities. A prime example is the introduction of *scoped storage* in Android 10, which fundamentally changed how apps interact with files. While intended to enhance user privacy by limiting broad access to shared storage, scoped storage has also presented developers with new challenges and potential vulnerabilities to address. However, despite its significance for user privacy and app functionality, no systematic studies have been performed to study Android's scoped storage at depth from a security perspective.**

In this paper, we present the first systematic security analysis of the scoped storage mechanism. To this end, we design and implement a testing tool, named ScopeVerif, that relies on differential analysis to uncover security issues and implementation inconsistencies in Android's storage. Specifically, ScopeVerif takes a list of security properties and checks if there are any file operations that violate any security properties defined in the official Android documentation. Additionally, we conduct a comprehensive analysis across different Android versions as well as a cross-OEM analysis to identify discrepancies in different implementations and their security implications.

Our study identifies both known and unknown issues of scoped storage. Our cross-version analysis highlights undocumented changes as well as partially fixed security loopholes across versions. Additionally, we discovered several vulnerabilities in scoped storage implementations by different OEMs. These vulnerabilities stem from deviations from the documented and correct behavior, which potentially poses security risks. The affected OEMs and Google have acknowledged our findings and offered us bug bounties in response.

I. Introduction

The security of data storage is a critical concern in modern mobile security. Unlike temporary data in memory or in transit, information in storage remains accessible for extended periods, prolonging the window of risk exposure. As mobile devices accumulate an ever-growing collection of sensitive data over time—ranging from personal information to financial records and proprietary business information—they become increasingly attractive targets for attackers. The challenge is further complicated in multi-user environments, where different apps share the same storage space, even sharing access to the same files, which requires a more fine-grained permission system to manage data access [1], [2], [3].

Previous research has exposed how attackers violate privacy and security by exploiting vulnerabilities in storage shared by apps. For example, Bianchi et al. [4] demonstrated the theft of authentication credentials via shared storage, and Dong et al. [5] revealed the use of shared storage as a covert channel for transferring user identifiers between apps, compromising user privacy. Additionally, Checkpoint [6] reported an attack where an untrusted application could maliciously replace a victim app's library files, leading to potential crashes or arbitrary code execution.

As a defense against shared storage vulnerabilities, Google introduced *scoped storage* in Android, a security model that aims to enhance protection for both apps and users by limiting access to shared storage spaces [7]. Scoped storage has shown promising improvements, with Lee et al. [8] estimating that scoped storage reduces attack operations—authorized operations that may be exploited by adversaries to escalate their privileges—by 54-71% in Android's external storage. Furthermore, Dong et al. [5] confirmed that when scoped storage is active, all malicious SDKs that violate user privacy by utilizing cross-app user identifiers are rendered inoperative.

Despite the benefits of scoped storage, it does not address all issues. For instance, Tuncay has shown that adversaries can escalate their privileges and obtain unauthorized access to all files via SDK downgrading on devices with active scoped storage [9]. Other anecdotal evidence [10] reveals that Android 12 had an implementation error, which led to a widely exploited loophole, despite scoped storage being active, resulting in violations of the security properties that scoped storage aims to achieve. Furthermore, while Android 13 partially addressed this issue, it was not until Android 14 that all known vulnerabilities associated with this loophole were fully resolved. These instances motivate us to explore three research questions: (1) Is Android's shared storage secure under all the security defenses, including scoped storage? (2) Does Android's storage implementation correctly comply with its specifications? (3) Given the recurrent over-the-air

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240340
www.ndss-symposium.org

updates across Android versions and OEMs, are there any inconsistencies in storage implementations, and if so, what are their security implications?

Studying these issues presents several challenges. First of all, although Android storage is now mostly governed by scoped storage, an app's ability to access a file is still determined by multiple distinct security rules and exceptions that are subtle and sometimes undocumented across different permissions, file types, and other variables. Hence, it remains unclear whether there are conflicts between these security rules, making the analysis even more challenging. Furthermore, Android supports multiple storage operation APIs, and while the expected results are clear, the correct implementations can vary and potential issues are unbounded, making it difficult to analyze the source code to identify potential problems. Additionally, various OEM implementations and Android versions exhibit different API behaviors across builds while expected to maintain the same standard of security and correctness. These versions coexist in the market and are actively used by users, making it challenging to develop scalable, automatic tools that are extensible to all available Android builds.

Prior work has examined Android storage and the impact of scoped storage. For example, PolyScope [11] is a tool designed to triage combinations of Android filesystem access control policies and identify potential logical vulnerabilities. However, this tool's capabilities are limited because it cannot identify implementation issues, and in fact, it did not reveal any feasible attacks that bypass the scoped storage defense when it is fully activated. Dong et al. [5] confirmed all SDKs identified as malicious become inoperable under scoped storage. While they were able to identify an attack for cross-app user tracking that works on fully activated scoped storage, it still requires the attacker to obtain runtime permissions to perform the attack.

In this paper, we propose ScopeVerif[1], a dynamic black-box testing tool for Android storage that is device-agnostic and can be used to identify arbitrary violations of given security goals. To avoid analyzing the large and complex Android codebase, we use dynamic analysis to systematically execute an array of file operations on a given Android device to reveal the operations that violate security properties. Specifically, we design ScopeVerif as a distributed dynamic testing tool where the controller runs on a PC and the workers are Android apps running on Android devices. The controller generates test cases, sends commands to worker apps to perform various file operations, collects feedback from the workers, and uses a violation oracle to determine if there are any security violations.

The violation oracle dynamically determines whether a file operation complies with specific security properties. It works by constructing two different environments: one where the security property is guaranteed to hold and another where the actual testing is performed. The violation oracle then conducts a differential analysis to determine if the tested file operation has the same outcome in both environments.

Using the violation oracle, ScopeVerif identifies violations of scoped storage rules. An analysis of similar efforts (see Section III-B) shows that ScopeVerif is the first testing tool capable of verifying Android's implementation of storage for arbitrary violations of a list of given security properties. Furthermore, by comparing the results across multiple versions and OEMs, ScopeVerif can reveal implementation discrepancies. To evaluate ScopeVerif and detect inconsistencies in scoped storage implementations, we ran the same experiments on multiple devices and across multiple Android versions.

Our study discovered 11 different types of violations, ranging from trivial cases, such as undocumented or unavailable features, to more security-critical violations, such as partially fixed loopholes or previously unknown privacy leakages. Google has acknowledged two of our findings: first, a high-severity loophole that allows unauthorized access despite scoped storage, and second, a secrecy violation that enables cross-app user identification without requiring any permissions. Additionally, Huawei acknowledged our finding of an integrity violation, which allows the creation of files within other apps' private folders. Both Google and Huawei offered us bug bounties for our findings.

In summary, we make the following contributions:

- **Formalize the properties of scoped storage.** We systematically studied the official Android documentation [13] and identified eight security properties that describe scoped storage's security guarantees.
- **Develop a black-box testing tool for Android's scope storage.** We developed ScopeVerif, the first automatic, black-box, and device-agnostic testing tool that targets Android storage. It verifies specified security properties and identifies violations of confidentiality, integrity, and availability. It also supports parallel testing on multiple devices from different OEMs and running various versions of Android.
- **Identify violations of scoped storage rules.** We identified 11 distinct issues, ranging from minor issues to more security-critical violations. Our findings were confirmed by Google and Huawei, both of which offered bug bounties.

II. Background

In this section, we explain the basics of Android storage and previously known attacks. In Section II-A, we detail how the Android storage model has evolved and describe the commonly used Storage APIs. In Section II-B, we discuss previously known attacks, including details such as how they work and what they can achieve.

A. Android Storage Basics

External Storage. In Android, a distinct filesystem partition is designated for various dynamically managed files, such as media and application updates, historically referred to as

[1]The source code of ScopeVerif can be found here [12].

the *External Storage* partition [14], [15]. This partition has different folders to separate public shared folders from private app-specific folders. The data stored in these public areas remains accessible and some persist even after the app that created them is uninstalled.

Prior to Android 10, Android's storage system used a coarse-grained access control approach, where read or write permissions were granted either for the entire external storage or not at all. For example, apps that needed to access their own folder had to request the `READ_EXTERNAL_STORAGE` permission. However, this permission also allowed them to access the private folders of other apps in external storage, forcing users into a situation where they must either grant access to everything or nothing. There was no way for users to grant an app access to only its own folder.

Scoped Storage. As a redesign of the original coarse-grained access control system, Android introduced the scoped storage feature starting with Android 10. Scoped storage uses Filesystem in Userspace (FUSE) [16] to redirect all file operations to the `MediaProvider` component for centralized access control. `MediaProvider` enforces scoped storage rules by unifying gating access to files: it can permit access, deny it, or provide a redacted version of it (i.e., with sensitive information removed).

Scoped storage ensures that each app has full access to its own app-specific folder without the need for any permission, where it remains completely isolated from the app-specific folders of other apps. Accessing other app's files in shared folders would require explicit user consent. The `READ_EXTERNAL_STORAGE` permission now only permits access to media files, further limiting potential misuse. Moreover, the `MANAGE_EXTERNAL_STORAGE` permission [17], which allows broader access to external storage, is strictly regulated on Google Play, ensuring that only apps with a valid need can request it.

Although Android 10 introduced scoped storage, it allowed apps to opt out of scoped storage and disable the defense at will [18], [9]. Since Android 11, apps could still opt out by targeting their SDK to Android 10 (API level 29) [19]. However, starting with Android 12, the system fully enforces scoped storage and ignores any requests to opt-out. Therefore, this paper focuses on Android versions starting with Android 12 up to the latest, Android 14.

Storage APIs. There are various methods to do file operations on Android's external storage. In general, an app can use a file path or URI to perform file operations. One of the commonly used APIs is the `File` API [20], which relies on a file path. The other methods rely on URIs, which usually require one API to get the URI, such as `MediaStore` [21] or Storage Access Framework (SAF) Picker [22]. Then, if the app wants to access the file, it will need to use another API such as `FileDescriptor` [23].

In Android 10, apps using scoped storage could not access files via direct file paths due to the effort required to intercept kernel calls. Starting with Android 11, scoped storage supports the File API, and access control is managed through `MediaProvider`, adhering to the same restrictions of scoped storage rules.

When using the `File` API to save a file where another file with the same name already exists, it will throw an exception. However, when using other URI-based APIs, the file-saving process will proceed, but the system will automatically increment the filename by adding a numeric suffix to avoid duplication (e.g., "(1)").

B. Known Storage Attacks

There have been several storage-related attacks with varying severity levels on Android [4], [5], [6]. Here, we will cover the ones that are most related to our work.

Account Hijacking. Apps often store login credentials so that users do not need to log in each time they use the app. However, if an attacker steals these credentials, they can use them on their own device to gain unauthorized access to the victim's account. Previous research [4] has shown that if apps store their login credentials in external storage, the credentials essentially become public information. This is because a malicious app can exploit the commonly granted `READ_EXTERNAL_STORAGE` permission to access them, given the lack of more fine-grained permissions.

Cross-app User Identifier. Within the same advertisement network, apps may seek to identify users in order to exchange information about users' preferences and deliver more personalized advertisements. Android mandates apps to use the *Advertising ID* for this purpose, while users can opt out or reset it at any time. However, previous research [5] shows that external storage can be used as a covert channel for apps to share custom user identifiers. Similar to an Advertising ID, these custom user identifiers may allow different apps installed on the same device to link various accounts to the same person. However, the user cannot opt out of this tracking or reset their identifier, which results in a breach of their privacy.

Squatting Attack. External storage is commonly used by apps to store software updates or dynamically loaded libraries. Since the permission `WRITE_EXTERNAL_STORAGE` is often granted, apps can write to external storage without restriction, including writing to the private folders of other apps. This allows them to create or modify files in the private folders of other apps and mislead the victim app into using the attacker-created file instead of their original files. According to a report by Checkpoint [6], attackers can maliciously substitute the library files of an app stored on external storage, leading to crashes or even arbitrary code execution in the victim app.

Downgrade Attack. A vulnerability discovered in Android's scoped storage implementation allowed apps to bypass storage access restrictions through SDK downgrading [9]. After obtaining scoped storage permissions, malicious apps could downgrade their target SDK level or modify storage settings, then update the app to gain unrestricted legacy storage access without requiring additional user permission. This effectively enables attackers escalate their storage privileges from the limited scoped storage access to full external storage access.

Starting with Android 12, this attack is no longer viable as apps are not allowed to opt out of scoped storage on devices offering it.

SAF Loophole. Since scoped storage, directories like `/Android/data` and `/Android/obb` are restricted. These directories contain apps' private folders, which are used to store app data, including large files like game assets. SAF loophole refers to a vulnerability that is commonly used by file manager apps like MiXplorer [10] to bypass scoped storage restrictions and access private app directories on Android devices. It leveraged SAF, which lets apps request access to specific directories.

This loophole has two known variations. Although Google blocked direct SAF access to the `/Android` directory in Android 12, the implementation did not block access to its subdirectories, allowing apps to still reach those files by requesting permissions at the subdirectory level. We refer to this issue as *SAF Loophole A*. Later, Google blocked `/Android/data` and `/Android/obb` in Android 13, but not their subdirectories, still allowing the SAF picker to directly access private app folders, such as `/Android/data/com.google.android.youtube`. We refer to this issue as *SAF Loophole B*.

III. Motivation

Since Android storage accumulates sensitive data over time, it has become an increasingly attractive target for attackers. The storage model is constantly changing across different versions and OEMs. With the introduction of scoped storage in Android 10, which set new security objectives, Android's storage model was completely redesigned, adding further complexity. This complexity raises three research questions:

1) How effectively does scoped storage ensure security and privacy?
2) How well do the existing implementations of this defense comply with the official Android documentation?
3) Are there inconsistencies due to the variations across OEMs and versions, and what are their security consequences?

Studies focusing on specific implementations can quickly become obsolete with each new design release. For this reason, it is essential to conduct a systematic, automated study that remains effective across various versions, OEMs, and security objectives. However, this presents several challenges.

A. Challenges

In this section, we outline three primary challenges in developing a testing tool that can effectively and efficiently verify different security properties across multiple Android versions and OEMs.

CH1: Large and Complex Code Base. One major challenge arises from the complexity and dispersion of the codebase. Ideally, access control checks regarding storage should be centralized and unified, but in reality, they are distributed across various concurrent API implementations in different components at different layers, from the lower layer's Linux kernel to the upper layer's Android system, written in different programming languages.

The diversity of implementation language and the dispersion of storage-related functionality throughout the source code complicate the manual analysis, as they require navigating an extensive amount of intertwined code. They also pose significant difficulties for automatic static-analysis tools since there can be potentially unbounded patterns of random coding errors, such as the SAF loophole [10], a known issue in scoped storage that allows apps to access other apps' private folders despite scoped storage being fully activated. While the standard access control is correctly implemented in `MediaProvider`, there is an additional component—the SAF picker—that separately enforces scoped storage rules and has an issue due to an incorrectly written regex pattern.

CH2: Differences across Android Versions and OEMs. Some parts of the Android codebase are not open-sourced. For instance, while the Android Open Source Project (AOSP) provides a publicly accessible code base, customized builds implemented by different OEMs or security fixes through Google Play system updates may be closed-source [24]. This lack of accessibility to source code complicates the compatibility of any approach that is based on white-box testing.

Additionally, Android updates and OEM customizations might include undocumented changes, making it difficult to maintain a consistent testing approach and identify crucial inconsistencies that could affect app security and functionality. For example, in the latest Android 14, Google no longer allows users to use the SAF picker to access an app's own private folder. This update is undocumented and differs from the implementations by Samsung or Huawei.

CH3: Existence of multiple storage-related APIs. A fundamental goal of our work is to automatically detect arbitrary security violations regarding file storage, as well as identify discrepancies between documentation and implementation in a precise way. Since there are many combinations of APIs for file operations, managing them can be challenging. This is especially true for URI-based APIs, where each viable combination may provide different feedback, creating many corner cases. Inconsistencies among different OEMs further complicate the issue, where different implementations do not necessarily mean they are insecure. For example, the File API throws an error if an app tries to create a file at a location where another app has already created a file with the same name. In contrast, MediaStore automatically renames the file by appending an index before the extension to avoid conflicts (e.g., `duplicate (1).txt`). Both file operations should be considered "working as intended," but defining rules to handle all such situations requires significant engineering work and is error prone.

B. Existing Solutions

To the best of our knowledge, no one has attempted to create a device-agnostic testing tool to automatically verify the implementation of scoped storage across different Android

versions and OEMs. However, there is existing work that studied scoped storage or Android access control policies in general. For instance, PolyScope [11] models scoped storage by extracting access policies from documentation for static analysis, which partially addresses **CH1**. However, it does not analyze the underlying Java implementations or dynamically verify vulnerabilities, limiting its ability to account for undocumented differences across Android versions and OEMs, and thus does not fully address **CH2**. Moreover, PolyScope is unable to detect unexpected implementation errors in storage-related APIs, such as the SAF loophole [10], leaving **CH3** unresolved. A detailed discussion of existing related work can be found in Section IX.

IV. ScopeVerif Design and Implementation

We explain the design and implementation of our black-box, device-agnostic dynamic testing tool for Android's scoped storage, named ScopeVerif. In doing so, we will also discuss how our design choices address the technical challenges discussed in Section III-A.

Overview. Our goal is to precisely identify security vulnerabilities as well as implementation inconsistencies with security implications in the various implementations of Android's storage APIs across different Android versions and OEMs. For this purpose, we first formalize the security rules for Android's storage based on the guidance provided in the official Android documentation (Section IV-A). Then, we translate these rules into concrete test cases that, when run, elicit specific sequences of API calls (Section IV-B). The generated test cases are tested using a dynamic analysis approach and a dedicated violation oracle, which utilizes differential analysis to reveal any violations of the tested security rules (Section IV-C). Finally, we analyze the results by clustering violations based on their root causes and compare the results across various Android versions and OEMs to identify implementation inconsistencies (Section IV-D). Figure 1 shows an overview of our approach. Additional implementation details can be found in Appendix B.

Our approach focuses on the results of file operations instead of analyzing the complex Android codebase, thereby addressing **CH1**. Additionally, it is device-agnostic and allows identifying violations across different file access APIs, Android versions, and OEMs, thereby addressing **CH2** and **CH3**.

A. Formalizing Rules from Documentation

For ScopeVerif to work consistently across different versions, we use a declarative approach in which a configurable list of rules that always need to hold is first used to generate test cases. These test cases are later used to guide dynamic analysis, and the results regarding their execution are checked by a violation oracle. This approach has two benefits. On the one hand, it allows us to consistently verify security properties across multiple Android versions and OEMs, addressing **CH2**. On the other hand, this approach also makes ScopeVerif future-proof, because it allows an operator to easily add new rules to be verified. A concrete example of how to add new rules is given in Section VIII.

We categorize Android storage security properties into the traditional CIA properties: confidentiality, integrity, and availability. Specifically, confidentiality ensures that the file cannot be accessed by unauthorized apps, integrity ensures that the file cannot be modified by unauthorized apps, and availability ensures that the file operations can be performed by authorized apps.

Based on the guidance provided in the official Android documentation [13], we formalized scoped storage into eight security rules. As detailed in Table I, to address **CH3**, we translate the documentation into security rules that describe security goals by defining the targeted path (i.e., private or shared folders), the types of action (e.g., read, write, move, etc.), the permission settings, and the types of security properties (i.e., confidentiality, integrity, or availability) the system wants to achieve.

To illustrate how a security rule from the documentation was formalized into a row in Table I, we take the rule `C1` as an example. The documentation [28] states: "[...] apps can no longer access files in any other app's dedicated, app-specific directory within external storage."

To translate this rule from natural language to a row in Table I, we first determine the type of the security rule. In the example, the documentation specifies that the apps' private files should be confidential, hence the type of the security rule is Confidentiality. The `ID` of rule `C1` starts with "C" because it is a Confidentiality rule. Similarly, "T" stands for an Integrity rule and "A" stands for an Availability rule.

An `Actions` set consists of file operation types. The documentation for `C1` does not mention any exceptions in the file operations, so the set of `Actions` includes all possible options: Create, Read, Overwrite, Delete, Move, and Rename (abbreviated as **`C, R, U, D, M, N`**). In contrast, the documentation for rule `A3` mentions that having the required permission should allow an app to read other apps' media files, except for location data. Since it only authorizes read operations, the set of `Actions` for `A3` is limited to **R** only.

Also, in case of `C1`, there are no exceptions for `File Attributes`, so this rule includes all file attributes, such as Content, Path, Size, Modified Date, Media Location, and Exceptions (abbreviated as *`CT`*, *`PT`*, *`SZ`*, *`MD`*, *`ML`*, *`EX`*). In contrast, for rule `C3`, the only `File Attribute` included is *ML*, as the documentation specifies that an app should not, without proper permissions, be allowed to access location data of another app's media file.

The set of `APIs` for `C1` includes `File`, `MediaStore`, and `SAF`, as rule `C1` applies to all APIs. The `Targets` for `C1` are `Other_Private` because the scope of this rule applies to the `Private` files of `Other` apps. In another case, `T1`, an app should not be able to modify another app's file in the `Download` collection without user consent (or the "all files access" permission). Thus, its `Targets` become

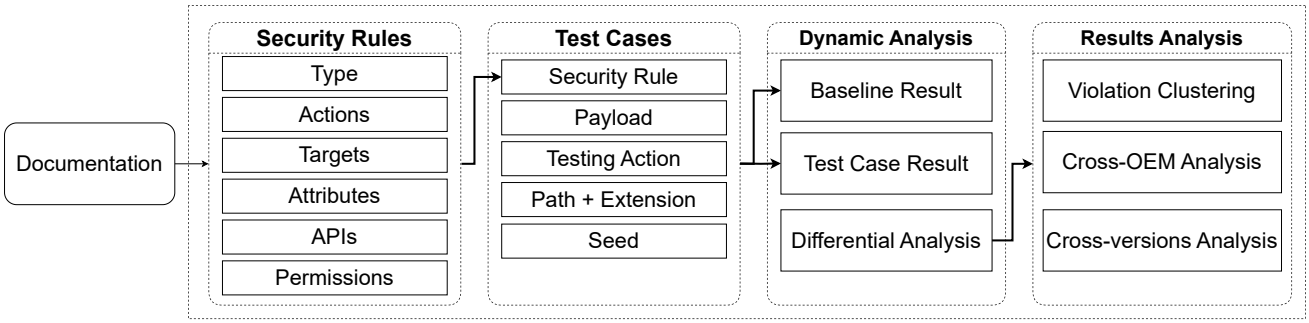

Fig. 1: Overview of ScopeVerif Approach.

TABLE I: Security Rules of Android Storage. A rule describes targeting whose file (Self or Other) and which collection (Media, Download, or Private), under what permission settings, using which APIs to perform what types of actions (Read, Write, etc.), and which attributes (Content, File path, Size, etc.) should hold which types of properties (Confidentiality, Integrity, Availability).

ID	Actions	File Attributes	APIs	Targets	Permissions
A1 [25]	**C**, **R**, **U**, **D**, **M**, **N**	*CT*, *PT*, *SZ*, *MD*, *ML*, *EX*	File, MediaStore, SAF	Self_Private	`None`
A2 [26]	**C**, **R**, **U**, **D**, **M**, **N**	*CT*, *PT*, *SZ*, *MD*, *ML*, *EX*	File, MediaStore, SAF	Self_Download	`None`
A3 [27]	**R**	*CT*, *PT*, *SZ*, *MD*, *EX*	File, MediaStore_SAF	Other_Media	`RES`
C1 [28]	**C**, **R**, **U**, **D**, **M**, **N**	*CT*, *PT*, *SZ*, *MD*, *ML*, *EX*	File, MediaStore, SAF	Other_Private	`AML`, `RES`, `WES`, `MES`, `WMS`
C2 [29]	**C**, **R**, **U**, **D**, **M**, **N**	*CT*, *PT*, *SZ*, *MD*, *ML*, *EX*	File, MediaStore	Other_Download	`AML`, `RES`, `WES`, `MES`, `WMS`
C3 [30]	**C**, **R**, **U**, **D**, **M**, **N**	*ML*	File, MediaStore, SAF	Other_Media	`RES`, `WES`, `MES`, `WMS`
T1 [31]	**C**, **U**, **D**, **M**, **N**	*CT*, *PT*, *SZ*, *MD*, *ML*, *EX*	File, MediaStore	Other_Download	`AML`, `RES`, `WES`, `WMS`
T2 [25]	**C**, **U**, **D**, **M**, **N**	*CT*, *PT*, *SZ*, *MD*, *ML*, *EX*	File, MediaStore, SAF	Other_Private	`AML`, `RES`, `WES`, `MES`, `WMS`

* The original documentation specifying each of the rules is cited next to the rule ID.
* The first letter of ID indicates the type of the security property: (C)confidentiality, In(T)egrity, (A)vailability.
* Actions: **C**: Create, **R**: Read, **U**: Overwrite, **D**: Delete, **M**: Move, **N**: Rename
* File Attributes: *CT*: Content, *PT*: Path, *SZ*: Size, *MD*: Modified Date, *ML*: Media Location, *EX*: Exceptions
* Permissions: `AML`: ACCESS_MEDIA_LOCATION, `RES`: READ_EXTERNAL_STORAGE, `WES`: WRITE_EXTERNAL_STORAGE, `MES`: MANAGE_EXTERNAL_STORAGE, `WMS`: WRITE_MEDIA_STORAGE, `None`: no permissions required.

`Other_Download`, and its `APIs` contain only `File` and `MediaStore`, because the usage of the `SAF` API requires user consent and, therefore, does not violate this rule.

As for the `Permissions`, the documentation does not specify any exceptions for `C1` regarding the permissions granted, therefore the set of `Permissions` is all storage-related permissions, including `AML`, `RES`, `WES`, `MES`, and `WMS` (detailed in Table I's caption), which means that regardless of the permissions granted, an app should not be able to access other apps' private files.

B. Translating Security Rules to Test Cases

After we formalize the security rules in Table I, we need to translate them into a set of test cases where each test case describes a sequence of API calls to be performed. Our goal is to generate test cases that enumerate all the possibilities of file-related operations that might violate the security rules to ensure the coverage of the search space, thereby addressing **CH1** and **CH3**. The generation should also be comprehensive and automatic in order to address **CH2**.

ScopeVerif generates test cases by iterating through all the possible attribute combinations for each security rule in Table I using Algorithm 1. Note that the procedure is generic and comprehensive, and it applies to all rules in Table I, allowing easy addition and modification of rules. In fact, if a new rule is added as a new row in Table I, the algorithm can generate test cases for the new rule automatically. Listing 1 and Algorithm 2 (in Appendix A) show a concrete example of how we internally represent a test case and demonstrate how ScopeVerif executes a test case.

C. Dynamic Analysis and Violation Oracle

We designed ScopeVerif to be a distributed dynamic testing tool. Its architecture is divided into two parts: the controller and the workers, as shown in Figure 2.

The controller runs on a PC and the worker apps are Android apps running on the tested Android device. The controller takes security rules as input and asks the `InputGenerator` to generate test cases based on the given rules. According to the generated test cases, the controller then sends commands to worker apps. The worker apps perform file operations on the filesystem based on the controller's

Algorithm 1 Test Case Generation

```
function GENERATETESTCASESFORALLRULES(rules)
    test_cases ← []
    for all rule in rules do
        for all api, action, target in VALIDCOMBINATIONS(rule) do
            test_cases.extend(GENERATETESTCASESFORONERULE(rule, api, action, target))
    return test_cases

function VALIDCOMBINATIONS(rule)
    valid_combinations ← []
    for all api in rule.apis do
        for all action in rule.actions do
            for all target in rule.targets do
                if api.is_valid_target(target) and api.is_valid_action(action) then
                    valid_combinations.append(api, action, target)
    return valid_combinations

function GENERATETESTCASESFORONERULE(rule, api, action, target)
    cases ← []
    for all path in target.get_paths() do
        /* Check if api/path/action combination is included in the specification (see Table I) */
        if rule.is_included(api, action, path) then
            for all ext in get_extensions_by_path(path.template) do
                for all perm in rule.permissions do
                    payloads ← GENERATE_PAYLOADS(api, target, max_payload_len)
                    for all payload in payloads do
                        case ← TestCase(rule, action, api, payload, perm, path.template, ext)
                        cases.append(case)
    return cases
```

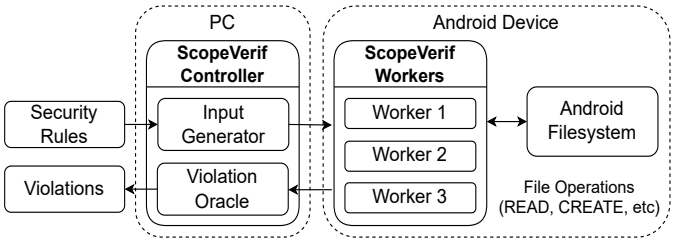

Fig. 2: Overview of ScopeVerif Architecture.

commands, collect results, and send feedback to the controller. Eventually, the controller asks the violation oracle to determine if there is a violation based on the workers' feedback. As each test case elicits a specific sequence of API calls, a violation oracle determines if a rule is violated based on dynamic execution results. Appendix A provides a step-by-step example of how ScopeVerif and the violation oracle operate. Additional implementation details are provided in Appendix B.

The design of the violation oracle is based on differential analysis. In general, the violation oracle runs each test case twice on the same device. The first time, it collects the baseline results, representing the expected outcome if the property holds. The second time, it collects the test case results. Any difference between the baseline results and the test case results indicates a violation of the tested property. Concretely, the violation oracle behaves differently depending on whether the tested rule is about confidentiality, integrity, or availability, as we will now explain.

Confidentiality. To construct the baseline for confidentiality, the violation oracle will execute a test case twice on the same device. The first time, it attempts accessing a non-existent file. The expected feedback, such as "No such file or directory", serves as the baseline result, indicating that confidentiality holds. Then, it executes the same test case a second time, but attempting to access a file that actually exists. If confidentiality holds, the two results should be exactly the same. In contrast, any difference indicates a confidentiality violation.

Integrity. For integrity, the violation oracle reads a file twice on the same device. The first time, it reads a specific file without any modification attempts. Then, the violation oracle executes the test case, allowing the attacker app to attempt modifying the file. Finally, the violation oracle reads the file a second time. Any difference in the results of the first and second read operations implies that the file was altered during the execution of the test case, indicating a violation of integrity.

Availability. The violation oracle uses two different users to execute the test case twice on the same device. The first user is root, who is capable of performing all file operations. The feedback from root executing the test case serves as the baseline. Then, the violation oracle executes the test case. If both results are the same, this indicates that the test case was able to perform the file operation specified by the test case. If the results differ, it implies a violation of availability, as the discrepancy indicates that the tested file operation failed.

D. Analysis of the Results

After the violation oracle determines whether a test case contains any violations of the tested security properties, ScopeVerif also attempts to cluster identified violations for further analysis. Note that identifying the root cause of a violation and grouping violations based on their root cause require manual investigation. Classifying a violation as a vulnerability involves subjective interpretation, as demonstrated by the varying responses we received from different OEMs. Ultimately, the final decision rests with the developers or organizations responsible, who apply their own criteria and risk assessments. For these reasons, we designed this part to be semi-automatic.

First, we select a subset of identified violations and manually investigate the root causes of the failed test cases. Using the insights from this investigation, we develop a script to automatically classify the violations in the subset into different types of security or privacy issues. Once the subset is fully analyzed and categorized, we apply the script to the entire set of test cases.

Finally, we review each group to identify patterns. This analysis allows us to precisely determine Android versions, devices, and OEMs affected (or not affected) by a specific issue. By following this approach, we gain better insights into the results provided by the ScopeVerif, which we will describe in the next section.

V. ScopeVerif Results

We designed our evaluation to answer the following questions:

- **Q1**: How efficient is the violation oracle in ScopeVerif at finding violations of security properties regarding scoped storage?
- **Q2**: How reliable is ScopeVerif when re-running experiments to find implementation inconsistencies across Android versions and OEMs?
- **Q3**: How long does it take for ScopeVerif to expose violations and inconsistencies?

We use the Pixel 3a for Android 12, and the Pixel 5a for both Android 13 and 14, all updated with the latest security updates and Google Play System Updates. Additionally, we use the Samsung Galaxy S22 with the latest build of Android 14 and the Huawei Mate 40 Pro running HarmonyOS 4, which is based on Android 14 and also enforces scoped storage. While the controller runs on a PC and is Python-based, it only sends ADB commands to the workers and collects results during the experiment. Therefore, the experiment's runtime is constrained by the workers.

In the rest of the paper, for brevity, we may refer to different OS implementations using the OEM's name that developed it and the Android version they utilized. For example, Google 12 represents Google Pixel's Android 12 implementation, whereas Huawei 14 represents Huawei's Android 14 version.

A. Finding Security Issues

To address Q1, we first used ScopeVerif to generate a search space of 22,217 test cases to verify eight security rules of scoped storage. As the search space was too large, we applied random sampling to explore it (see Appendix B for implementation details). This heuristic sampling technique is based on a hypothesis that assumes security or privacy issues are clustered in the search space instead of uniformly scattered. If the hypothesis holds true, extensive exploration of the search space is not strictly necessary to find most issues, and the sampling method should be sufficient to identify them.

To test this hypothesis, we verify a small sample of 505 test cases, which is 2.27% coverage of the space, on a Pixel 5a running Android 14, and then scale up to a larger sample of 2,501 test cases, which is 11.26% of the search space. We evaluate how many security issues can be found in the small experiment, as well as assess how many new issues are uncovered in the larger experiment. If there is a proportional increase in identified issues in the larger experiment, that indicates the need for exploring the whole search space.

For the smaller experiment tested on Android 14, ScopeVerif found 88 violations in total, which means 17.4% of the tested cases. Compared to the larger experiment, ScopeVerif found 509 violations in total, which is 20.35% of the tested cases. ScopeVerif then classifies these violations (see the details in Section IV-D) as actual security issues. We found seven issues in both the smaller and larger experiments. This means that by randomly exploring approximately 2.27% of the search space, ScopeVerif identified the same number of security issues as would be found in approximately 11.26% of the search space.

Table II shows the results of ScopeVerif for the smaller experiment. Specifically, it shows the types of issues ScopeVerif detected (e.g., Squatting Attack, SAF Loophole, etc.) based on our semi-automatic analysis and the number of test cases where ScopeVerif discovered a specific issue to be viable.

Combined with our evaluation of cross-version analysis and cross-OEM analysis (see Section V-B), ScopeVerif identified 10 distinct issues in total, each of which is presented as a row in Table II. These issues are found in both shared and private storage, encompassing various known and unknown violations of different security properties. For example, ScopeVerif identified confidentiality violations, such as the Metadata Leak, EXIF Leak, and Download Leak, which could lead to privacy issues or misunderstandings for developers and users.

It also found the well-known issue, the SAF loophole [10], including variants (See Section II) SAF Loophole A and SAF Loophole B in Google 12, and found SAF Loophole B in Google 13. These issues allow an attacker to completely bypass scoped storage, violating the confidentiality, integrity, and availability of other apps' files in both shared and private storage. Additionally, ScopeVerif uncovered availability violations, including the Squatting Attack, EXIF Failure, MediaStore & File Failure, SAF Auto-rename, SAF Restrictions,

and Other Failures, which could cause crashes or unexpected usability issues. In Section VI, we will discuss the details and explanations of these issues in more depth.

After a thorough evaluation, we determined that nine out of these 10 issues were previously unknown, and two of them were either security or privacy issues. We reported the privacy issues due to the Metadata leak violation to Google and the security issue, the new variant SAF Loophole C to Huawei. Both companies acknowledged the reports and offered us bug bounties. In addition, we observe that Samsung, like Huawei, also has seven violations of SAF Loophole C (details in Section VI-B). However, after communicating with Samsung, we confirmed that applying the latest Google Play System Update to the operating system resolved the issue in Samsung 14.

Answer to Q1: ScopeVerif found 10 issues, nine of which were previously unknown, including two security or privacy concerns.

TABLE II: Number of violations detected by ScopeVerif for each tested device and Android version.

Issues	Google 12	Google 13	Google 14	Samsung 14	Huawei 14
Metadata Leak	15	8	8	10	8
EXIF Failure	6	0	0	0	4
Download Leak	1	1	1	1	1
SAF Loophole	125	102	0	7	7
Squatting Attack	1	1	1	1	1
EXIF Leak	2	10	10	10	2
MediaStore & File Fail.	17	17	17	17	19
SAF Auto-rename	2	2	2	2	3
SAF Restrictions	0	0	49	41	44
Other Failures	0	0	0	0	1
Test Cases in Violation	169	141	88	89	90
Total Test Cases	505	505	505	505	505

* The column names represent which Android versions are running on which devices. For example, Google 14 means Android 14 running on a Google device (i.e., Pixel). Similarly, Samsung 14 represents a Samsung device running Samsung's version of Android 14.

B. Identifying Inconsistencies

Beyond finding security issues, our analysis aims to understand the inconsistencies in different OS versions and how OEMs introduce mistakes. These inconsistencies, which refer to variations in mistakes across each OEM and version, can expose unique patterns that may lead to security issues. To answer Q2, we conduct the same experiment on Google Pixel devices running Android 12, 13, and 14, as well as on Samsung and Huawei devices running Android 14. We then compare the results to identify inconsistencies and manually analyze the root cause of inconsistencies. Notice that by comparing the results between Google 12, Google 13, and Google 14, we focus on inconsistencies between Android versions while keeping the device constant. On the other hand, by comparing the results between Google 14, Samsung 14, and Huawei 14," we keep the Android version constant at the latest version.

As shown in Table II, out of the 10 issues, only three remain consistent across all implementations. ScopeVerif not only produces statistics for each issue but also shows how these statistics change from version to version and how they differ between OEMs. On the one hand, our analysis indicates that violations have decreased over time as newer OS versions address issues such as the SAF Loophole, while we also identified the introduction of new issues, such as SAF restrictions. On the other hand, inconsistencies between Google's devices and OEM devices underscore potential problems. For instance, SAF Loophole C was discovered by comparing SAF Loophole violations across Google 14, Samsung 14, and Huawei 14. Despite all running Android 14, the seven additional violations observed on Samsung 14 and Huawei 14 compared to Google 14 suggest the presence of unique issues specific to these OEM implementations. More details on the findings of these experiments will be discussed in Section VI.

Answer to Q2: ScopeVerif found that seven out of 10 issues had inconsistent implementations, revealing bug fixes or new issues.

C. Discovery Efficiency

For Q3, we record the running time for each test case and report the maximum, minimum, and average test times across different Android versions and devices. As shown in Figure 3, the average time to test a single case on Android 14 with the Pixel 5a is approximately 28 seconds, indicating that exploring the entire search space would require approximately 7.2 days. However, based on the answer to Q1 (See Section VI), the findings of ScopeVerif converge relatively quickly. Exploring a small sample of 505 cases yields the same results as 2,501 cases, identifying 10 issues, nine of which were previously unknown, while taking around only four hours. Additionally, since ScopeVerif can run on multiple devices simultaneously, using five devices would allow us to identify all inconsistencies in the same 4-hour period.

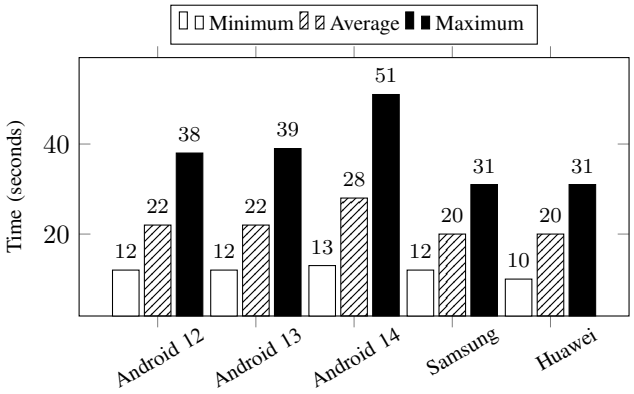

Fig. 3: Runtime of Test Cases

Answer to Q3: ScopeVerif can identify previously unknown security issues within a day.

VI. Identified Security Issues

In this section, we discuss the security issues identified by ScopeVerif, listed in Table II.

A. Issues Found in Shared Storage

Despite the enhanced privacy measures of fully activated scoped storage, ScopeVerif found previously unknown issues that could lead to privacy violations. Surprisingly, these vulnerabilities might even make it easier for attackers to compromise user privacy compared to the older storage model. Furthermore, ScopeVerif identified availability issues that could allow an attacker to crash another app.

Metadata Leak. Scoped storage restricts apps from accessing other apps' files. However, ScopeVerif found that while the content of the files is not accessible, the meta information, such as the existence, size, and modified date of a file might still be accessible without needing any permissions.

ScopeVerif first identified that using the `File` API, by calling `exists()` directly, allows an attacker to determine if another file exists. Similarly, `size()` and `lastModified()` are not blocked by scoped storage defenses and return correct results as usual. Additionally, ScopeVerif found that calling `readText()` can also leak the existence of a file. Specifically, when a file exists, even if the attacker cannot access its content, the exception would contain `EACCES (Permission denied)`, while accessing a non-existent file contains `ENOENT (No such file or directory)` instead.

The `MediaStore` and SAF picker also leak file existence in shared folders due to their auto-rename mechanisms. Under scoped storage, files created by other apps are by default invisible to an app. However, if a file is created in the same directory with the same filename, both APIs will automatically rename the saved files by adding an index such as "(1)". Therefore, simply checking if the file created was renamed leaks to an attacker whether a file with a given name already existed.

While `MediaStore` correctly adds the index before the extension, the SAF picker incorrectly appends the index after the extension. This is identified by the verifier as the SAF auto-rename issue. Specifically, instead of `my_file (1).txt`, the SAF picker creates "my_file.txt (1)," which changes the file's extension and could potentially cause unexpected usability issues.

The Metadata Leak issue was found in Android 12, affecting both app-specific folders and shared folders. It was partially fixed in Android 13, which addressed all leaks in app-specific folders. However, as of today, even in the latest Android 14, the Metadata Leak issue remains unresolved in shared folders.

As prior research has shown, cross-app user identifiers can be established using external storage as a covert channel [5]. Unfortunately, scoped storage does not mitigate this issue. In fact, it inadvertently facilitates this attack by enabling attackers to exploit the Metadata Leak vulnerability in shared storage.

Before scoped storage, apps would need the runtime permissions `READ_EXTERNAL_STORAGE` to read files from shared storage and `WRITE_EXTERNAL_STORAGE` to write files to shared storage. While scoped storage restricts an app's interaction with other apps' files, it actually reduces the friction for an app to manage its own files. More specifically, an app can create files in shared storage without any permissions and read its own contributed files without any permissions.

Together with the Metadata Leak bug, an app can write to shared storage without any permission, while another app can read one bit of information (i.e., existence or non-existence of a file) from the file created by the first app. This essentially establishes a covert channel that is fully accessible to any apps installed on the device, allowing malicious apps to freely communicate custom user identifiers without the user giving any consent.

For example, assume apps in the same advertisement network called "bad" predetermine a user identifier length of 20 bits. App A from the "bad" network installed on the device generates a 20-bit user identifier, encodes it into binary form, and then creates files in shared storage (in this case, at most 20 files) corresponding to the encoded identifier (e.g., creates a hidden file `.bad_4` if the 4th bit is 1, and does nothing if it is 0). Later, App B, as a member of the "bad" network, gets installed on the victim's phone. App B checks for the existence of all 20 possible files (e.g., `.bad_1`, `.bad_2`, etc.). App B decodes the 20 bits of data into a unique identifier number that identifies the user.

In May 2022, we reported the Metadata Leak issue to Google, highlighting the risks of cross-app user identification. Google responded that it was "not a security vulnerability" and was "working as intended." In April 2024, we reported the same issue again, explaining the same risks. This time, Google acknowledged the issue but marked our report as a duplicate of another bug report. We suspect this change is due to the evolving threat model of Android. Initially, cross-app user identification was not seen as a security concern. Later, as Google's perspective shifted, they began to take this issue more seriously, leading to someone else's report being acknowledged.

Squatting Attack. Previous squatting attacks worked by replacing a file to force the victim app to use the attacker-provided file. This exact attack no longer works, but a variation still works under the scoped storage defense. Instead of an integrity violation, it now results in an availability violation, which could potentially crash the victim app. This is accomplished by creating a file in shared storage before the victim app attempts to do so.

Since the file created by the attacker is meant to be invisible to the victim app, the app cannot detect if another file with the same name already exists without exploiting the Metadata Leak bug. Furthermore, the file created by the attacker is intended to be inaccessible to the victim app without required permissions or user consent. Any attempt by the victim app to write to this file would result in an exception.

Normally, using the SAF picker or MediaStore, the API would automatically rename the file if another with the same name already exists. However, this is not the case with the File API. While the File API does not automatically rename the file, attempting to write a file owned by another app would cause an exception. If not handled properly, this could lead to a crash of the victim app.

The root cause of this issue is the same as the Metadata Leak, as file name conflicts can occur in shared storage when other apps' files are invisible. When the conflicts happen intentionally, a Metadata Leak occurs and confidentiality is violated. When the conflicts happen unintentionally, a Squatting Attack could occur and availability could be violated. To avoid duplicate reporting, we included the details of the root cause of the Squatting Attack in the same report that we submitted in May 2022.

B. Issues Found in App-specific Storage

Most issues found by ScopeVerif for app-specific storage are caused by the SAF picker, including SAF Loophole A, and SAF Loophole B. ScopeVerif also discovered two new variants, SAF Loophole C and SAF Loophole D, in OEM implementations. Even after applying the latest security update to Android 14, we were still able to find another variant SAF Loophole D with the help of ScopeVerif.

SAF Loophole A and B. The SAF loophole allows apps to access other apps' private folders and fully bypass scoped storage. This is due to a non-centralized implementation of access control for scoped storage. As explained in Section II-A, file operations should be redirected to the `MediaProvider` for centralized access control, but there are exceptions. While the standard component (i.e., `MediaProvider`) implements most of the restrictions for scoped storage, there are additional components that require implementing the duplicated rules. In this case, Android developers inadvertently failed to implement scoped storage restrictions in several components of the SAF picker, such as the `ActionHandler` [32]. ScopeVerif not only identified previously known issues but also found new ones.

ScopeVerif also highlighted how SAF loopholes are changing across Android versions. There were 125 violations in Android 12 because the SAF picker completely failed to block access to other apps' private folders, including SAF Loophole A and SAF Loophole B. Both SAF Loophole A and SAF Loophole B are previously known and reported. These issues were partially fixed in Android 13, as shown in Table II, where the number of violations dropped from 125 to 102, with only SAF Loophole B issues remaining.

The 102 SAF Loophole violations left in Google 13 are due to a regex pattern that is incorrectly written and does not block the path that directly accesses another app's private folder. Therefore, if an attacker knows the package name of another app, they can still predict the folder path and use the SAF picker to access that app's private folder.

Eventually, all known issues of the SAF Loophole were fixed in Android 14, as the violation count shows 0. While Google had trouble implementing this correctly and needed several major versions to do so, ScopeVerif identified this problem within a day.

SAF Loophole C. When checking OEM implementations, ScopeVerif found that both Samsung and Huawei had an inconsistency compared to Google's Android 14. This inconsistency allowed the creation of files in other apps' private folders using the SAF picker, while blocking all other types of file operations such as reading, deletion, etc. This issue is another variant of the SAF Loophole, but different from A and B, so we call it SAF Loophole C. SAF Loophole C could lead to a squatting attack, where an attacker creates a malicious file and lures the victim to use the attacker-provided file, as explained in Section II.

In May 2024, we reported these issues to both Samsung and Huawei. After communication with Samsung, we confirmed that the issue in Samsung was silently fixed after applying the latest Google Play System Updates. On the other hand, Huawei acknowledged our findings, issued a bug ID (HWPSIRT-2024-02103), and offered us a bug bounty.

SAF Loophole D. During our analysis, we discovered another issue in scoped storage that ScopeVerif did not initially identify. Due to the limitations explained in Section VIII, this issue was highlighted by our tool, but not in a fully automated way, as it required manual analysis for complete and reliable reproduction. For this reason, it is not listed in Table II for evaluation.

Similar to SAF loopholes, this issue allows a malicious app to create files in other apps' private folders even in the latest Android 14. This problem arises due to the default behavior of the SAF picker, which accesses the last accessed folder if no viable path is provided or if none of the specified paths can be accessed. This access is privileged and not blocked by scoped storage. Consequently, even with the latest Google Play System Update, which fully fixes all known SAF loopholes (i.e., the variants A, B, and C) in Android 14, if a malicious app had used the SAF picker to access another app's private folder before the security update was applied, it could retain access to the last accessed private folder after the update is installed.

This could lead to two potential risks. By checking whether the created files have been renamed or not, a malicious app could exploit this bug to reveal the existence of another file in another app's private folder on external storage, leading to privacy leakage. Another risk is that the attacker might perform a squatting attack by creating a file before the victim app does, luring the victim app to use the attacker-provided file, which may cause it to crash or become vulnerable to a fraud attack.

Since this issue is caused by AOSP code, in July 2024, we reported it directly to Google, who acknowledged it as a *high-severity* vulnerability and offered us a bug bounty.

C. Documentation Discrepancies

Aside from the security and privacy issues identified in shared storage (Section VI-A) and private storage (Section VI-B) that attackers could exploit, we also discovered that some violations reported by ScopeVerif are not exploitable. These non-exploitable violations are caused by undocumented features or policy exceptions. While these discrepancies are not insecure in themselves, they might create misunderstandings for both developers and users.

EXIF Leak and EXIF Failure. Media files, such as photos, can contain EXIF information, including location data. According to documentation [30], an app must declare `ACCESS_MEDIA_LOCATION` permission in the manifest file and request it during runtime to read this location data, obtaining user consent. However, ScopeVerif found an exception, EXIF Leak, where `MANAGE_EXTERNAL_STORAGE` can also provide access to unredacted media files, which contradicts the documentation. Since `MANAGE_EXTERNAL_STORAGE` is a special permission requiring stricter vetting from the Google Play Store and more complex user approval, it is assumed that obtaining this permission allows bypassing restrictions from weaker permissions. However, this remains an issue as it may lead developers and users to falsely believe that media files can only be accessed with the `ACCESS_MEDIA_LOCATION` permission.

This issue was initially found in Android 12 with only two violations but later increased to 10. This increase was due to the fixing of another issue, EXIF Failure, where using the SAF picker to access a file in Android 12 would always strip location data regardless of app permissions, making it always "confidential" but violating availability. In Android 13, after the EXIF Failure was resolved, the number of EXIF Leak violations became 10. Note that this special case simply shifted an availability issue to a confidentiality issue, and no problematic test cases were missed from our results due to this reason.

Download Leak. While documentation [29] states that accessing another app's file in the `Download` folder mandates user consent (i.e., SAF picker), ScopeVerif found that an exception exists where `MANAGE_EXTERNAL_STORAGE` also allows an app to access other apps' files in `Download`. However, this is also an undocumented exception where `MANAGE_EXTERNAL_STORAGE` practically implies full access to shared storage. Therefore, it is not considered a security issue by Google, similar to the EXIF Leak.

SAF Restrictions. During our cross-version analysis, ScopeVerif discovered a silent update in Android 14 restricting apps from using the SAF picker to access their own files in app-specific folders for privacy protection. This contradicts the documentation [25], which states that apps should have read and write access to their own files without permissions under scoped storage. This undocumented change, absent in earlier Android 14 builds but later silently added to the latest Android 14, could cause apps to fail unexpectedly and lead to usability issues. With further investigation, we realized that this issue was caused by a software update to Android 14 that silently changed how the SAF picker blocks file access. Also, ScopeVerif identified inconsistencies between Samsung and Huawei compared to Google, where OEMs' Android still allows creating files in an app's folder using the SAF picker, which causes further complications.

MediaStore & File Failure. Since ScopeVerif extensively explores a wide range of possible API usages, it can uncover exceptions and edge cases not addressed by the documentation. For example, it discovered that if a non-JPG file is created using the `File` API, the `MediaStore` API cannot create, overwrite, or delete that file, leading to an availability violation. As highlighted in Table II, this violation leads to 17 MediaStore & File Failure violations for every Google and Samsung device tested. During cross-OEM analysis, ScopeVerif discovered that JPG files also fail to work on Huawei devices, resulting in two additional violations and bringing the total number of MediaStore & File Failure violations for Huawei devices to 19. To the best of our knowledge, this is not a documented restriction.

VII. Discussion

We discuss three potential causes that contribute to the issues identified in Section VI. For each potential cause, we also propose recommendations to mitigate the issue.

A. Spread-out Codebase of Scoped Storage

There are multiple methods to access external storage, for example, `File` API, `MediaStore` API, native system call `open`, or using system-provided components like SAF picker. Ideally, by using Filesystem in Userspace (FUSE) [16], all file operations in user space will be redirected to the `MediaProvider` component. Then, the `MediaProvider` component enforces the rules of scoped storage by centralizing file access control. Depending on the situation, it can grant access, block it, or return a redacted version of the file with sensitive data removed. However, in reality, not all access is handled by FUSE.

For example, scoped storage prohibits any API from accessing private app folders, as shown by security rules C1 and T2 in Table I. To enforce these rules, Android uses a regex pattern within components like `MediaProvider` to prevent unauthorized access to private folders. However, SAF Loophole A was identified in Android 12, revealing that the regex pattern was flawed—it only matched `/Android`, but not `/Android/data` or `/Android/obb`. In Android 13, SAF Loophole B emerges as the fixes only blocked `/Android/data` and `/Android/obb` without covering their subdirectories. Addressing this oversight required significant modifications. Multiple commits were made to correct this issue, such as [32], [33], [34], [35], amending the duplicated regex pattern that was spread across various components, such as `MediaProvider`, `ExternalProvider`, and `ActionHandler`.

Even in the same component, the access control might not be centralized. The SAF Loophole D issue, which we previously explained in Section VI-B, is one example. When using the SAF picker, it determines if the `initialUri` should be allowed according to scoped storage rules, so it calls the method `shouldPreemptivelyRestrictRequestedInitialUri(initialUri)`, which contains the regex pattern discussed above to filter out illegal access to other apps' private folders. If such access is blocked, it will not proceed to `launchToDocument(initialUri)` to provide access. After the access is blocked, it proceeds to access the last accessed folder instead by calling `initLoadLastAccessedStack()`. Since this function does not have any regex filtering, it allows an attacker to bypass the scoped storage limitation. If an attacker previously used the SAF picker to access other apps' private folders before the SAF Loophole was fixed, then their last accessed stack is the private folder of another app. Since there is no restriction implemented in accessing the last accessed stack, the attacker's unauthorized access would remain effective even after applying the security update which fixes the known issues of SAF.

While the initial design of scoped storage is to use a single component `MediaProvider` to gate access for all the file operations, the actual implementation, as we observed, indicates the scoped storage-related codebase is much more spread out than expected, which we suspect to be the root cause for all of the coding issues found by ScopeVerif that lead to security or privacy issues in app-specific folders. In order to fix this issue, we suggest refactoring the codebase of scoped storage to have unified gating access. This will be easier for developers to maintain, as a single issue in one part of the codebase will not automatically lead to multiple issues in various components requiring fixes. It will also be easier for testing, as a centralized codebase makes automatic analysis of the source code much easier.

B. Conflicting Requirements of Scoped Storage

Scoped storage aims to allow apps to contribute files to shared storage without restrictions while also preventing them from interfering with other apps' files in shared storage. However, these two requirements are logically conflicting and therefore infeasible to achieve if apps physically share the same folder. We believe this is the root cause of most privacy issues found in shared storage by the verifier.

In the current storage scheme, when an app writes a file to a shared folder, a conflict naturally arises if there is a file with the same name already present. In fact, when there is a name conflict, the API will either automatically rename the newly created file or throw an exception, both of which inevitably reveal the existence of another app's file.

To achieve the same goal, we suggest having virtually shared folders. In this case, apps will have their own designated sharing folders where they put files to be shared with other apps, making it impossible to have naming conflicts Then, when apps with the required permissions want to access the files of other apps, the operating system could scan the sharing folders all apps and display all files in a virtual folder. In this implementation, it is fine for two apps to create files with the same name in the same shared folder. Selecting a shared file in this virtual shared storage could be done using the file path along with its owner.

C. Unclear Documentation of Storage APIs

Other issues ScopeVerif found are all due to discrepancies between documentation and implementations. While these issues do not directly lead to security or privacy issues, they might cause unexpected usability issues or misunderstandings.

For example, Download Leak in Table II is caused by unclear documentation. While the documentation claims accessing files in the Download collection mandates user consent via the SAF picker (T1 in Table I), there is an undocumented exception: the `MANAGE_EXTERNAL_STORAGE` permission. In scoped storage, the `MANAGE_EXTERNAL_STORAGE` permission is highly regulated and implies full access to shared folders in external storage. Therefore, it was not considered as a security issue.

However, developers unfamiliar with the details of scoped storage might mistakenly believe their files in the Download collection are protected unless access is explicitly granted via the SAF picker. In fact, apps with the `MANAGE_EXTERNAL_STORAGE` permission, such as file backup applications, can also access these files. To prevent any misunderstanding, we recommend having unified documentation that explains how various Storage APIs and permissions interact within scoped storage and explicitly outlines all exceptions.

VIII. Limitations and Future Work

Adding new rules. ScopeVerif is extensible since it allows specifying new security rules to be verified. As long as new security rules can be described in terms of CIA properties, they can be verified using the existing design of the violation oracle.

For example, if future Android versions no longer allow apps to write to shared storage (except in the Downloads folder), we can simply add two rules to the ones listed in Table I: (1) When writing to the Downloads folder, Availability must hold. In this case (referring to Table I), `ID` could be A4 and `Actions` is **C**, **D**, **U**, **N**. `Attributes` and `APIs` both enumerate all available options. `Targets` is `Self_Download` and `Permissions` is `None`. (2) When writing to other folders in shared storage, Integrity must hold. This would require another rule. In this case, `ID` could be T3 and `Targets` is `Other_Media`. For all other columns, the values are the same as A4. After adding these rules, the violation oracle can automatically verify rule compliance without implementing ad-hoc checks.

However, ScopeVerif can currently only verify rules that can be described in terms of confidentiality, integrity, or availability. For instance, if a new rule for Non-Repudiation is introduced, requiring the Android system to always track

who last edited a file, ScopeVerif cannot currently verify this property. Future work can extend our approach to use differential analysis, constructing baselines accordingly, and extending the violation oracle to verify additional types of security or privacy properties.

Another limitation arises when security rules involve newly introduced Android APIs. In this case, it may be necessary to update ScopeVerif (in particular, Algorithm 1) to enumerate the usage of new APIs during test case generation. The worker apps may need modifications as well to perform file operations using these new APIs. However, in this case, the core component of ScopeVerif (i.e., the violation oracle) does not require updates.

Scope Limited to File Operations. ScopeVerif is designed to generate and verify test cases consisting exclusively of file operations (e.g., create, delete, copy, rename). As such, it cannot detect vulnerabilities, such as the Downgrade Attack discussed in Section II, which require non-file operations, such as app updates or SDK version changes. Furthermore, since ScopeVerif focuses on verifying the enforcement of scoped storage when fully activated, scenarios where apps opt out of scoped storage are considered out of scope. The minimum Android version examined by ScopeVerif is Android 12, where opting out of scoped storage is no longer possible.

This scope limitation is a deliberate design choice to emphasize the security properties of the active scoped storage implementation. Future work could extend ScopeVerif test case generation to include non-file operations by modifying the controller app's enumeration logic and implementing corresponding functionality in the worker apps.

Violation Oracle Assumptions. Based on the methodology of ScopeVerif, the conclusion of ScopeVerif is valid only if the control result retains the property. Given that ScopeVerif uses dynamic analysis, controlling variables can be complex. Currently, ScopeVerif resets storage between test cases without reinstalling apps or resetting the device to maintain efficiency. However, if certain test cases permanently alter the state of the operating system, the initial state for subsequent tests may be compromised. As a result, the security property in the control result might be violated, leading to incorrect conclusions by the verifier.

This limitation is evident in the issue SAF Loophole D, discussed in Section VII-A, which causes non-deterministic behavior and has been acknowledged by Google. In this issue, the SAF Picker records an app's Last Accessed Stack and relaunches to the same stack if the given `initialUri` is blocked. ScopeVerif does not reset the SAF Picker state, which persists even after reboot and system updates. Without access control for the last accessed stack, this creates an abnormal state where an attacker can always create files in the last accessed folder using the SAF Picker. This issue initially caused false positives and negatives in our analysis, which we resolved by manually adding explicit handling for this corner case. The latest version of our tool is no longer affected by this problem, and reproducing the results now requires no manual effort to remove false positives.

Nevertheless, this issue highlighted a current limitation of our violation oracle: it must start from a normal device state to detect any abnormal state triggered by a test case. As future work, we can improve our oracle to be less dependent on such assumption, or we may find better ways to efficiently reset a device's state between each test case.

Fuzzing Heuristics. Currently, our testing tool generates the entire search space and randomly selects a set of test cases for evaluation. It does not rely on any heuristics to guide more efficient sampling based on known results. We conducted an exploration using existing experimental results to train a model that predicts the likelihood of unexplored test cases resulting in violations. The initial results are promising, showing an increase in the violation ratio relative to the total number of tested cases. However, the number of bugs identified does not consistently increase and, in some instances, decreases. Thus, we found that random sampling identifies no fewer bugs than our heuristic-based methods. In future work, we intend to explore methods for selecting test cases more effectively, particularly with respect to uncovering new security issues rather than merely counting violations.

Emerging Mobile Operating Systems. While this work focuses on Android's scoped storage enforcement, future research could extend these methodologies to other emerging mobile operating systems, such as Huawei's Harmony-NEXT, which represents Huawei's shift away from Android to its proprietary platform. As these systems evolve, it will be important to analyze their storage models and security frameworks to identify potential vulnerabilities and assess compliance with security properties like confidentiality, integrity, and availability.

IX. Related Work

A. Storage Vulnerabilities on Android

Previous research has shown that Android storage is vulnerable due to its coarse-grained access control mechanisms. Bianchi et al. [4] demonstrated that authentication credentials could be stolen via shared storage, allowing unauthorized attackers to bypass authentication and access victim accounts. Reardon et al. [36] and Dong et al. [5] revealed that shared storage can be exploited to transfer user identifiers between apps, bypassing Android's permission system and compromising user privacy. Liu et al. [37] discovered that shared storage on Android can be used to leak users' phone numbers, locations, and other sensitive information. Gisdakis et al. [38] used data mining and machine learning techniques to infer user gender, age, race, and other sensitive information from photos and voice messages stored in shared storage. Not only privacy, but also data integrity is threatened, as shown by Du et al. [39], who found that files stored in shared storage can be corrupted, potentially leading to voice message hijacking, installing malicious apps, or even phishing attacks. Additionally, Tuncay has shown that adversaries can obtain unauthorized access to all files by downgrading their app to use legacy storage (i.e., by opting out of scoped storage or

by downgrading their target SDK level to 28) after having obtained the storage permission on a device with active scoped storage [9].

Most of these studies were either conducted before the introduction of scoped storage or they did not evaluate its security impact. In contrast, our work explicitly considers scoped storage and systematically uncovers issues that persist despite its introduction.

B. Access Control Design for Android's Storage

Ahmad et al. [40] compared the storage access mechanisms between Android and iOS and found that, while iOS always requires user interaction each time an app accesses files from other apps, Android employs a persistent coarse-grained access control mechanism that allows apps with the appropriate permissions to maintain access to all files in shared storage.

This behavior is the fundamental reason for most vulnerabilities discussed above, stemming from the different access control mechanisms between Android and iOS. To address these issues, a line of work has proposed fine-grained access control models for Android's shared storage [41], [1]. These models introduce file-level granularity to Android's access control system, providing users with more control over file access and enhancing data isolation between applications. The latest file access control model (i.e., scoped storage) has adopted some of these ideas, allowing for fine-grained access control policies for shared storage. However, despite the fine-grained permission control model in scoped storage, we identified design flaws in its access policies. These flaws could lead to potential cross-app user identification without requiring any permission.

C. Automated Analysis of Access Control Policies

It is important to evaluate the implementation of fine-grained permission control mechanisms. However, as discussed in Section III-B, existing studies have limitations in fully addressing all research challenges. PolyScope [11] triages the combinations of access control policies of Android storage, but it does not consider the underlying Java code, nor does it dynamically confirm identified issues, as discussed in Section III-B. ACMiner [42] and AceDroid [43] assess the actual implementation, but do not focus on evaluating file-access permission checks, which depend on a combination of checks in different codebases, including the Linux kernel, FUSE [16], SAF picker, and Media Provider. Their analysis is constrained by limited code coverage due to challenges in decompilation and runtime modifications, and are fragile to differences in Android versions, OEMs, and APIs. They also do not build baselines directly from documentation, leading to false positives and requiring manual intervention for resolution.

X. Conclusion

This paper presents a systematic security analysis of the scoped storage defense. In particular, we implemented a dynamic testing tool, ScopeVerif, to verify the correctness of Android storage implementations as well as studying the inconsistencies between Android versions and devices. ScopeVerif found 10 distinct issues, nine out of which were previously unknown, including two security or privacy concerns. We reported these issues to Google and OEMs. Both Google and Huawei offered us bug bounties for our findings. Finally, we proposed improvements to the current scoped storage design, implementations, and documentation.

XI. Acknowledgments

This work was supported by Google's ASPIRE funding program. The views and opinions expressed in this paper are those of the authors only and do not necessarily reflect the views or positions of the funding agencies. We thank Haining Chen, René Mayrhofer, Dave Kleidermacher, and Roxanna Aliabadi Walker for their feedback on this paper.

References

[1] Feiqiao Huang, Wenjia Wu, Ming Yang, and Junzhou Luo. A Fine-Grained Permission Control Mechanism for External Storage of Android. In *Proceedings of the 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2911–2916. IEEE, 2016.

[2] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 3–14, 2012.

[3] Güliz Seray Tuncay, Soteris Demetriou, and Carl A Gunter. Draco: A System for Uniform and Fine-Grained Access Control for Web Code on Android. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[4] Antonio Bianchi, Eric Gustafson, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Exploitation and Mitigation of Authentication Schemes Based on Device-Public Information. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pages 16–27, 2017.

[5] Zikan Dong, Tianming Liu, Jiapeng Deng, Haoyu Wang, Li Li, Minghui Yang, Meng Wang, Guosheng Xu, and Guoai Xu. Exploring Covert Third-Party Identifiers Through External Storage in the Android New Era. Prepublication at https://www.usenix.org/system/files/sec24summer-prepub-442-dong.pdf, 2024.

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

[7] Scoped Storage on Android. https://developer.android.com/about/versions/11/privacy/storage.

[8] Yu-Tsung Lee, Haining Chen, William Enck, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Giuseppe Petracca, and Trent Jaeger. Polyscope: Multi-policy Access Control Analysis to Triage Android Scoped Storage. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[9] Güliz Seray Tuncay. Android Permissions: Evolution, Attacks, and Best Practices. *IEEE Security & Privacy*, 2024.

[10] Mishaal Rahman. Android 13 Makes File Managers Less Useful by Fixing a Loophole. Esper Blog, https://www.esper.io/blog/android-dessert-bites-28-file-manager-loophole-closed-73891524, 2022.

[11] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, 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 (USENIX Security 21)*, pages 2579–2596, 2021.

[12] ScopeVerif. https://github.com/purseclab/ScopeVerif.

[13] Google. Android Developer Guides. https://developer.android.com/guide, 2024.

[14] Demystifying Internal vs External Storage in Modern Android. https://tdcolvin.medium.com/demystifying-internal-vs-external-storage-in-modern-android-c9c31cb8eeec.

[15] René Mayrhofer, JV Stoep, Chad Brubaker, Dianne Hackborn, Bram Bonné, Güliz Seray Tuncay, Roger Piqueras Jover, and Michael A Specter. The Android platform security model (2023). *arXiv*, 2023.

[16] Use Scoped Storage with FUSE. https://source.android.com/docs/core/storage/scoped#using-scoped-storage-with-fuse.

[17] Request All-Files Access. https://developer.android.com/training/data-storage/manage-all-files#all-files-access.

[18] Opt Out in Your Production App. https://developer.android.com/training/data-storage/use-cases#opt-out-in-production-app.

[19] Storage Updates in Android 11. https://developer.android.com/about/versions/11/privacy/storage#scoped-storage.

[20] File — Android Developers. https://developer.android.com/reference/java/io/File.

[21] MediaStore — Android Developers. https://developer.android.com/reference/android/provider/MediaStore.

[22] Open Files Using the Storage Access Framework. https://developer.android.com/guide/topics/providers/document-provider.

[23] FileDescriptor — Android Developers. https://developer.android.com/reference/java/io/FileDescriptor.

[24] Abbas Acar, Güliz Seray Tuncay, Esteban Luques, Harun Oz, Ahmet Aris, and Selcuk Uluagac. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2024.

[25] App Access Restrictions. https://source.android.com/docs/core/storage/scoped#app-access-restrictions.

[26] Access Your Own Media Files. https://developer.android.com/training/data-storage/shared/media#media_store, 2023.

[27] Access Other Apps' Media Files. https://developer.android.com/training/data-storage/shared/media#access-other-apps-files.

[28] Access to App-Specific Directories on External Storage. https://developer.android.com/about/versions/11/privacy/storage#other-app-specific-dirs.

[29] Storage Access Framework Required for Accessing Other Apps' Downloads. https://developer.android.com/training/data-storage/shared/media#saf-other-apps-downloads.

[30] Media Location Permission. https://developer.android.com/training/data-storage/shared/media#media-location-permission.

[31] Update Other Apps' Media Files. https://developer.android.com/training/data-storage/shared/media#update-other-apps-files.

[32] Commit: 070e145. https://android.googlesource.com/platform/packages/apps/DocumentsUI/+/070e14547db2e03590e295ac25b76cfb9f45fc78.

[33] Commit: 1e78acf. https://android.googlesource.com/platform/packages/providers/MediaProvider/+/1e78acfaa147d89f2bbbd9803582738dc0ca10a8.

[34] Commit: fe79a43. https://android.googlesource.com/platform/packages/providers/MediaProvider/+/fe79a43a890d9c54655b0ad0beeab58958aa1cfb.

[35] Commit: 7f5667b. https://android.googlesource.com/platform/frameworks/base/+/4af5db76f25348849252e0b8a08f4a517ef842b7.

[36] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pages 603–620, 2019.

[37] Xiangyu Liu, Wenrui Diao, Zhe Zhou, Zhou Li, and Kehuan Zhang. Gateless Treasure: How to Get Sensitive Information from Unprotected External Storage on Android Phones. *CoRR*, abs/1407.5410, 2014.

[38] Stylianos Gisdakis, Thanassis Giannetsos, and Panos Papadimitratos. Android Privacy C(R)ache: Reading Your External Storage and Sensors for Fun and Profit. In *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*, pages 1–10, 2016.

[39] Shaoyong Du, Pengxiong Zhu, Jingyu Hua, Zhiyun Qian, Zhao Zhang, Xiaoyu Chen, and Sheng Zhong. An Empirical Analysis of Hazardous Uses of Android Shared Storage. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[40] Mohd Shahdi Ahmad, Nur Emyra Musa, Rathidevi Nadarajah, Rosilah Hassan, and Nor Effendy Othman. Comparison Between Android and iOS Operating System in Terms of Security. In *Proceedings of the 8th International Conference on Information Technology in Asia (CITA)*, pages 1–4. IEEE, 2013.

[41] Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. Enforcing File System Permissions on Android External Storage: Android File System Permissions (AFP) Prototype and OwnCloud. In *Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 949–954. IEEE, 2014.

[42] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 25–36, 2019.

[43] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[44] Media Store. https://developer.android.com/training/data-storage/shared/media#storage-permission-not-always-needed, 2023.

APPENDIX

A. Example: Generate and Execute a Test Case for Rule C1

In this section, we first explain what a test case consists of, then we explain how ScopeVerif concretely generates such a test case from a security rule. Finally, we demonstrate how ScopeVerif executes the test case and identifies a violation.

Definition. In general, a test case consists of seven key attributes:

- `Rule`: The security rule being tested (e.g., `C1` for Confidentiality).
- `Storage Location`: The location where the file is stored (e.g., a private folder of another app).
- `File Type`: The type of file involved (e.g., PDF, image).
- `API`: The API used for file operations (e.g., the `File` API, the `MediaStore` API).
- `Permissions`: The permissions granted to the app executing the test case.
- `File Operation`: The main operation being tested (e.g., Move, Read).
- `Operation Sequence`: Additional operations performed before or after the main operation to test the rule under various conditions.

Generation. To generate test cases, Algorithm 1 starts by enumerating all applicable combinations from rule configurations, such as `APIs`, `Actions`, and `Targets`. Then, it generates multiple test cases by further enumerating all possible `Operation Sequences` (i.e., "payloads").

An `Operation Sequence` involves adding additional file operations executed either before or after the main `File Operation`. The purpose of these sequences is to explore whether the security rule still holds under various conditions. For example, the test might include writing to the file before reading it or moving it before reading it to ensure that the tested security rule remains enforced throughout different sequences of operations.

For each test case, Algorithm 1 sets the `Rule` of the test case to the security rule being tested. In the cases relative to `C1`, Algorithm 1 sets it to "Confidentiality," expecting no information leakage from the accessed file. ScopeVerif also determines the `Storage Location` from its `Targets`—the private folders of other apps. Because scoped storage mandates

Algorithm 2 Confidentiality Test Case Execution

```
function ExecuteConfidentialityTestCase(test_case)
    /* Set permissions for all apps */
    for all app in apps do
        if app ≠ gamma then
            permissions ← default_settings
        else
            permissions ← test_case.permissions
        setPermissions(app, permissions)
    /* Assign attacker and victim based on scope */
    if checkScope(test_case.path, Scope.Self) then
        /* If the target has a scope of "Self," the test case will be executed on the attacker's own files. */
        attacker ← gamma
        victim ← gamma
    else
        /* If the target has a scope of "Other," the test case will be executed on the victim's files. */
        attacker ← gamma
        victim ← alpha
    /* Collecting Baselines */
    result_on_nonexisting_file ← attacker.run(test_case, test_case.final_action)
    /* Execute the payload */
    for all action in test_case.payload do
        if action.type == "SETUP" then
            /* Perform setup action */
            victim.create_file(test_case)
        else
            /* Execute malicious action as attacker */
            attacker.run(test_case, action)
    /* Execute final action and obtain results */
    result_on_existing_file ← attacker.run(test_case, test_case.final_action)
    /* Compare Results */
    violation ← is_different(result_on_non_existing_file, result_on_existing_file)
    return violation
```

that certain folders store specific file types (e.g., `DCIM` and `Pictures` store only pictures) [44], it deduces the `File Type` based on the `Storage Location`.

Execution. Listing 1 shows an example of a test case, relative to the security rule `C1`. For this specific example, we will detail how ScopeVerif executes it and identifies a previously unknown violation. In general, ScopeVerif controls three worker apps installed on the testing device: Alpha, Beta, and Gamma. For the test case in the example, ScopeVerif first assigns app Gamma as the attacker and app Alpha as the victim. This test case describes moving a PDF file from Alpha's app-specific directory in external storage to Gamma's folder using the `File` API. To execute the test case, ScopeVerif runs Algorithm 2 using the test case from Listing 1. In summary, as a result, ScopeVerif instructs Gamma and Alpha to perform the following steps:

1) Set proper permissions for all apps
2) Move file from Alpha to Gamma
3) Read file from Alpha to collect baseline result
4) Create file in Alpha
5) Move file from Alpha to Gamma again to collect the test case results

After executing the above steps, ScopeVerif collects the following results:

1) **Result on Non-Existing File (Baseline):**
   - **Target:** PDF file in Alpha's private folder
   - **Action:** Move File
   - **Outcome:** "No Such File" Exception
2) **Result on Existing File (Test Case):**
   - **Target:** PDF file in Alpha's private folder
   - **Action:** Move File
   - **Outcome:** "Permission Denied" Exception

ScopeVerif (and, in particular, its violation oracle component) then compares the results collected. In particular, in the example, the baseline result (executed on a non-existing file) shows "No Such File" Exception, while the test case result shows a "Permission Denied" Exception. The difference between the two results indicates that the app has violated rule `C1`, meaning that the test case has shown how it is possible to leak the existence of a file to an unauthorized app. We refer to this issue as Metadata Leak (we discussed it in Section VI-A).

B. Additional Implementation Details of ScopeVerif

In this section, we provide more technical details of the implementation for how the controller generates test cases and how worker apps execute test cases.

Generating Test Cases in InputGenerator. After we have a list of security rules, ScopeVerif will explore the search space as much as possible, looking for potential violations. To define

Listing 1 A Test Case Generated for Rule C1

```
rule_id: "C1",
final_action: Move,
api: File,
path: "sdcard/Android/data/",
extension: ".pdf",
payload: {
  Read, File,
  Setup, File },
permissions: {
  ACCESS_MEDIA_LOCATION,
  READ_EXTERNAL_STORAGE,
  MANAGE_EXTERNAL_STORAGE,
  WRITE_EXTERNAL_STORAGE,
  WRITE_MEDIA_STORAGE }
```

the search space, `InputGenerator` enumerates all combinations of variables from security rules to generate test cases, as detailed in Algorithm 1. As discussed in Section IV-B, each security rule has different attributes, and these attributes will be enumerated and translated into a set of test cases. Note that while most attributes can be exhaustively enumerated, we opt for a heuristic approach that only enumerates common values to avoid excessive computation. For example, we only enumerate {`".pdf"`, `".txt"`} when enumerating different file types to access in the `Downloads` collection, instead of enumerating all file types in the real world.

In order to fairly explore each security rule, we applied a weighted random sampling technique to `InputGenerator`. For example, `A1` has a search space of 3,650 cases, while `T1` has a search space of 832 cases. If we want to test a total of 200 cases in `A1` and `C1`, we might explore the `T1` space more thoroughly but not as thoroughly in the `A1` space. To solve this, we ensure each security tule has the same percentage tested by our tool instead of the same number of cases. In this example, ScopeVerif will explore 163 cases in `A1` and 37 cases in `T1`.

In a test case, `InputGenerator` would generate a "payload," which is a sequence of actions we want the attacker app to perform. We generate all possibilities for the attacker's actions, given an upper limit of length k. While Android storage is influenced by various components, ranging from the low-level codebase to the applications' codebase, it does not exhibit many deep levels of internal states within the Android storage mode. Additionally, historical attacks on Android storage have never required a complex sequence of actions to be triggered. Therefore, based on our understanding and observations from previously known issues, we opted for a heuristic limit of 2.

Consequently, the attacker is limited to performing only two actions before the victim app performs its action. For example, in a traditional squatting attack, where the attacker attempts to replace a file that will be used by the victim app, the payload would involve a single "write" action to the victim app's file, followed by a "read" action performed by the victim. In this case, the payload length is one, which means this test case will be included in our experiments.

Worker Apps. All the test cases generated will eventually be executed by worker apps on devices running Android. However, worker apps have no knowledge of security rules or test cases. They only take commands, execute them, and return feedback. To ensure worker apps can communicate with each other, ScopeVerif requires ADB debugging to be enabled on the device.

In order to allow the violation oracle to conduct differential analysis (See Section IV-C), ScopeVerif has three worker apps in total. Before the experiments, the violation oracle assigns different roles to these workers, based on the security rules that the ScopeVerif is testing. To be more specific, both the confidentiality experiment and the integrity experiment would need at most two workers when an attacker attempts to access or modify another app's private files. Availability experiments would need at most three workers because, in certain test cases, there could be an app that has the targeted resource, a victim app that is trying to access the resource, and another attacker app that is attempting to prevent the victim from accessing the resource.

In order to control variables while efficiently testing the security properties, we use several techniques. For example, we reset the state of Android storage to ensure each experiment starts from a deterministic state. Since we are only testing storage-related properties, a full app reinstall is not necessary. Instead, before each experiment, we delete all files in the shared storage of external storage and app-specific folders in external storage.

Another technique to control variables is to perform string replacement in comparison results. For example, when testing the availability of file creation, while both the control and comparison results might contain feedback indicating successful creation, their metadata, such as modified time, could be different. In this case, we predefine all naturally different but irrelevant variables for each experiment and replace the strings in the comparison result to make them the same as in the control result. By doing this, we rule out false positives when detecting bugs using the violation oracle.

In the confidentiality and integrity experiments, control results are produced by timing: confidentiality control results are collected before the creation of the tested file to ensure zero information, guaranteeing absolute confidentiality, while integrity control results are collected before any modifications, ensuring absolute integrity. In contrast, the availability experiment requires control results collected with full capabilities and permissions, not achievable by timing alone. Instead, the violation oracle uses `shell` privileges to perform the action and collect the control result. ScopeVerif does not require the device to be rooted and uses `shell` user privileges to produce the control result.