A-SIT

Zentrum für sichere Informationstechnologie - Austria

Analyse der Sicherheit und Performance von Scoped Storage unter Android

A-SIT

Analyse der Sicherheit und Performance von Scoped Storage unter Android

Autor:
Florian Draschbacher:
florian.draschbacher@iaik.tugraz.at

Datum: 31.08.2022

Abstract/Zusammenfassung:
Im Laufe der letzten Jahre hat sich Android immer mehr vom traditionellen desktop-ähnlichen Paradigma eines gemeinsamen Dateisystems entfernt und stattdessen die Trennung von Daten verschiedener Anwendungen immer mehr forciert. Generell können Anwendungen nun auf öffentlichen Speicher nur noch über die Storage Access Framework (SAF) oder MediaStore APIs zugreifen. Während für diese Restriktionen bisher noch Workarounds bestanden, hat Android 11 die Situation noch deutlich verschärft. Das Release brachte eine transparente Übersetzungsschicht der alten File-API auf den MediaStore. Obwohl die Existenz dieses neuen Systems nur sehr spärlich dokumentiert wurde, sind dessen drastische Performance-Einbußen sogar für Endnutzer deutlich spürbar.

In diesem Projekt wird ein ausführlicher Überblick über die verschiedenen Aspekte der unter modernen Android-Versionen verfügbaren Schnittstellen für den Dateizugriff geboten. Neben der Analyse der Sicherheitsaspekte der Implementierung werden in einem Benchmark die offiziellen Performance-Angaben von Google überprüft.

Inhalt

1. Einleitung

Wurden Handheld-Computer zu Beginn als verkleinerte, weniger leistungsstarke Version eines Desktop-Computers positioniert, so haben sich mit der Zeit eigenständige Paradigmen entwickelt, die die besonderen Möglichkeiten von mobilen Plattformen berücksichtigen. In der öffentlichen Diskussion stehen bei solchen Betrachtungen üblicherweise die Benutzeroberflächen von mobilen Anwendungen im Vordergrund, doch auch tieferliegende Schichten moderner Mobil-Betriebssysteme weichen von den älteren Desktop-Pendants ab.

Spätestens seit der Einführung des iPhones hat sich im Mobil-Bereich das Konzept des Sandboxings etabliert. Hier wird jeder Applikation am mobilen Gerät nur ein genau definierter Bereich des Dateisystems zugestanden, in das Dateien abgelegt werden dürfen. Für den Austausch von Dateien über Applikations-Grenzen hinweg stehen lediglich Schnittstellen zur Verfügung, die vom Betriebssystem kontrolliert werden. Transaktionen müssen in den meisten Fällen vom Nutzer explizit angestoßen oder erlaubt werden.

Während sich Apple mit iOS von Anfang an klar für den exklusiven Einsatz von Sandboxing entschieden hat, ging Google mit dem Konkurrenzsystem Android deutlich konservativer vor. Hier war lediglich eine (zu Beginn vom System automatisch zugestandene) Erlaubnis nötig, um auf den Großteil des Dateisystems zugreifen zu können. Zwar wurde jedem Programm ein Ordner zugewiesen, auf den andere Anwendungen keinen Zugriff hatten, dennoch legte eine Vielzahl an Anwendungen ihre teils vertraulichen Daten im öffentlichen (gemeinsamen) Speicherbereich ab.

Um diesem Problem beizukommen, wurden Schritt für Schritt kleinere Veränderungen am Betriebssystem vorgenommen, die das Android-System weiter vom Desktop-Paradigma eines gemeinsamen Dateisystem wegrückten. Mit Android 10 wurde schließlich Scoped Storage eingeführt. Anwendungen haben nun Schreibzugriff auf den öffentlichen Speicher, ohne eine entsprechende Erlaubnis zu benötigen. Lediglich der Lesezugriff auf Dateien, die von anderen Anwendungen angelegt wurden, bedarf einer Erlaubnis durch den Nutzer. War das Scoped-Storage-System unter Android 10 noch optional, so wird es Anwendungen unter Android 11 nun aufgezwungen. Es ist nicht mehr möglich, ohne stichhaltige Begründung Zugriff auf das gesamte Dateisystem zu erhalten. Begründungen müssen vor der Veröffentlichung einer Anwendung an Google übermittelt werden.

Aus Nutzersicht ist diese Änderung unter zwei Aspekten interessant: Erstens ergeben sich durch die Änderung Verbesserungen in der Gesamtsituation der Datensicherheit am Gerät. Zweitens ist relevant, wie sich die neue Lösung auf die Performance von Anwendungen auswirkt. Beide Aspekte sollen in diesem Bericht beleuchtet werden.

2. Hintergrund

In diesem Abschnitt sollen jene Technologien erklärt werden, die für das weitere Verständnis der späteren Ausführungen notwendig sind.

2.1. Android-Anwendungen

Anwendungen für das Android-System werden in der Regel in Java verfasst. Nach dem Kompilieren liegt die Anwendung als APK-Datei vor, die auf Geräten installiert werden kann. Schlüsselbestandteil einer Android-Anwendung ist die Datei AndroidManifest.xml, die zur Deklaration aller Interaktionspunkte der App mit dem Betriebssystem (oder anderen installierten Apps) dient. Deklariert werden können hier unter anderem Activities (Benutzeroberflächen-Bildschirme), Services (Im Hintergrund laufende Applikations-Elemente ohne Benutzeroberfläche) oder ContentProvider (siehe unten).

2.2. Content Provider

Eigentlicher (und ursprünglich vorgesehener) Zweck von Content Providern ist die Möglichkeit, Datenbanken mit anderen Anwendungen zu teilen. Innerhalb des Android-Systems wird diese Möglichkeit etwa vom Contacts-Provider genutzt, um Kontakt-Daten aus dem Adressbuch systemweit zur Verfügung zu stellen. Im Laufe der Entwicklung des Android-Betriebssystems zeichnete sich allerdings ab, dass sich die ContentProvider-Schnittstelle auch dazu eignet, Dateien zur Verfügung zu stellen.

Jeder von einer App über einen ContentProvider bereitgestellte Datenpunkt (ursprünglich eine Zeile einer Datenbank) wird dabei systemweit über eine einzigartige URL identifiziert. Andere Anwendungen können dann über die ContentResolver-Schnittstelle des Systems die URL einer Ressource auflösen, um so einen Datenbank-Cursor (Möglichkeit zum Auslesen von Ausschnitten einer Datenbank) oder einen Datenstrom (InputStream oder OutputStream) zu erhalten. Über letztere Möglichkeit ist es möglich, nicht nur Dateien zum Lesen bereit zu stellen, sondern auch Dateiinhalte von anderen Apps schreiben zu lassen.

2.3. Discretionary Access Control

Hierbei handelt es sich um einen grundlegenden Sicherheitsmechanismus im Linux-Kernel, auf den das Android-Betriebssystem aufbaut. Bei entsprechender Unterstützung des zugrundeliegenden Dateisystems wird für jeden Ordner und jede Datei hinterlegt, welche Benutzer Schreib- oder Leseberechtigung haben. Unter Android wird dieses System dazu benutzt, installierte Anwendungen voneinander abzukapseln. Jeder Anwendung wird ein eigener Benutzer (Linux User ID = UID) zugewiesen.

3. Die Entwicklung des Android-Dateisystems

In diesem Abschnitt werden die Veränderungen nachvollzogen, die in den verschiedenen Android-Iterationen am Dateisystem vorgenommen wurden.

3.1. Android 1.0 (2008)

Schon seit der ersten Android-Version sieht das System für Anwendungen eine Unterscheidung zwischen öffentlichem und privatem Speicher vor. Zusätzlich gibt es eine Unterscheidung zwischen internem und externem Speicher.

Die Unterscheidung zwischen öffentlichem und privatem Speicher ist konzeptionell simpel: Es handelt sich bei privatem Speicher um einen Ordner, auf den nur eine Anwendung (und das Betriebssystem) zugreifen kann. Die Zugriffsrechte werden über Discretionary Access Control (DAC) kontrolliert. Der private Speicher kann im internen oder externen Speicher liegen.

Beim internen Speicher handelt es sich in jedem Fall um ein Speicherelement, das physisch im Android-Gerät verbaut ist und daher nicht entfernt werden kann. Die Bezeichnung externer Speicher war ursprünglich für ein Speicherelement gedacht, das nicht fix verbaut ist (etwa Speicherkarten). Schon bald bezeichnete es aber einfach einen fix im Gerät verbauten Speicherbereich, der über mehr Kapazität verfügt als der interne Speicherbereich. Zur Unterscheidung zwischen diesem Bereich und echten Speicherkarten setzte sich für Letztere schließlich die Bezeichnung Removable Storage durch (in der Android-Dokumentation wird auch Secondary External Storage oder Portable External Storage verwendet).

In Anlehnung an das kurz zuvor vorgestellte iOS-Betriebssystem (damals noch iPhoneOS bezeichnet), versuchte Google ebenfalls von Beginn an, Anwendungen zum Sandboxing zu motivieren und empfahl, für die Interaktion mit dem Dateisystem ausschließlich den privaten Speicher zu nutzen. Alle Dateien, die mit anderen Apps geteilt werden mussten, sollten über ContentProvider bereitgestellt werden [1]. Um allerdings den Anwendungsentwicklern auf der noch jungen Plattform zu ermöglichen, mithilfe von bestehenden Komponenten aus dem Java-Ökosystem schnell Ideen zu realisieren, wurde die Java File-API ebenfalls unterstützt. Für den Zugriff auf Speicherbereiche abseits des privaten App-Ordners musste keine besondere Berechtigung angefordert werden. So setzten sich vorerst Paradigmen durch, die von Desktop-Systemen übernommen worden waren.

Von Beginn an stand auch ein vom System bereit gestellter ContentProvider namens Media Store zur Verfügung. Dieser sollte als zentraler Index für alle Medien-Dateien (Videos, Musik, Fotos) am Gerät dienen. Media-Player-Apps, besonders etwa die vorinstallierte Bilder-Galerie konnten so auf eine vorindizierte Liste aller verfügbaren Medien zurückgreifen. Der Index wurde vom System automatisch im Hintergrund aktuell gehalten. Anwendungen konnten lediglich einen neuen Scan anstoßen, aber keine Dateien zum Index hinzufügen.

3.2. Android 1.5 (2009)

Der externe Speicher wurde nun über eine neue WRITE_EXTERNAL_STORAGE-Permission geschützt. Diese musste allerdings lediglich im Android-Manifest angefordert werden und wurde vom System automatisch gewährt, ohne dass der Benutzer in die Entscheidung eingebunden war. Weiterhin durften alle Anwendungen ohne besondere Berechtigung vom externen Speicher lesen.

3.3. Android 2.2 (2010)

Jeder App wurde nun ein eigener Ordner im externen Speicher zugewiesen. Dort konnten Dateien abgelegt werden, die im app-privaten Ordner keinen Platz hatten. Weiterhin befand sich der app-private Ordner im internen Speicher, der üblicherweise nur über wenig Kapazität verfügte. Obwohl jeder App auf Basis ihres Paketnamens ein eigener Ordner zugewiesen wurde, war der Zugriff keineswegs auf diese App

beschränkt. Jede App mit der WRITE_EXTERNAL_STORAGE-Berechtigung konnte in den app-spezifischen Ordner anderer Apps schreiben. Lesbar war der Ordner ohne jegliche Berechtigung [2].

3.4. Android 3.0 (2011)

Der Media Store ContentProvider wurde um eine Files-Tabelle erweitert [3]. Bisher wurden nur Bilder, Videos und Musik indiziert. Die Änderungen waren notwendig geworden, um das bestehende Indizierungs-System auch für den neu eingeführten USB-Datenaustausch via MTP nutzen zu können.

3.5. Android 4.4 (2013)

Mit Android 4.4 begannen Googles ernsthafte Bemühungen, die File-Sandbox konsequent durchzusetzen. Für den Lese-Zugriff auf externen Speicher wurde ab nun eine READ_EXTERNAL_STORAGE-Permission benötigt (in WRITE_EXTERNAL_STORAGE implizit enthalten). Dafür war es nun möglich, Dateien ohne besondere Berechtigung im app-spezifischen Ordner auf externem Speicher abzulegen. Apps mit READ_EXTERNAL_STORAGE-Permission konnten allerdings weiterhin von diesem externen app-spezifischen Ordner lesen.

Apps durften nun auch nicht mehr auf Removable Storage schreiben (außer die Ziel-Datei befindet sich in einem eigens für die App erstellten Ordner).

Die größte Änderung in Android 4.4. kann aber in der Einführung des *Storage Access Frameworks* gesehen werden. Damit war es nun möglich, den Nutzer einzelne Dateien oder Ordner-Strukturen auswählen zu lassen, auf die eine App fortan Lese- bzw. Schreibzugriff erhalten sollte. Wurde die Zugriffskontrolle bisher auf Basis von DAC entschieden, so gab es nun erstmals dynamische Entscheidungen, die vom Nutzer gesteuert wurden.

Zusätzlich wurde die DocumentProvider-Infrastruktur angelegt, über die es möglich war, Datenquellen wie Cloud-Speicher in das Storage Access Framework einzubinden.

3.6. Android 7.0 (2016)

Mit Android Nougat wurde das Konzept des Storage-Access-Framework auf Removable Storage ausgedehnt [4]. Der Nutzer konnte nun den Schreibzugriff auf Bereiche eines USB-Sticks oder einer SD-Karte gewähren. Vorerst handelte es sich allerdings um eine API, die von der SAF-API entkoppelt war.

3.7. Android 8.0 (2017)

Android 8.0 verbesserte einzelne Aspekte des Storage Access Frameworks. Erstmals wurde deutlich, dass durch die schrittweise Umstellung auf das SAF wohl mit Performance-Einbußen zu rechnen war. Es wurde ein Mapping zwischen Media Store URIs und DocumentProvider URIs eingeführt, um Medien-Apps die langsame Durchquerung von Scoped-Directory-Bäumen zu ersparen [5].

3.8. Android 10 (2019)

Mit Android 10 wurde die komplette Abkehr vom traditionellen öffentlichen Dateisystem angekündigt. Ab nun wurde von Scoped Storage gesprochen: Apps erhielten ohne besondere Berechtigung Zugriff auf den externen Speicher, aber nur auf einzelne Bereiche: Auf den app-spezifischen Ordner (wie schon in Android 4.4) und auf von der App selbst erstellte Einträge im Media Store. Die READ_EXTERNAL_STORAGE-Permission erlaubte nur noch den Zugriff auf Einträge im Media Store, die

von anderen Apps angelegt wurden. Für den Schreibzugriff auf Mediendateien anderer Apps reicht die WRITE_EXTERNAL_STORAGE-Permission nicht mehr aus, es muss explizit die Erlaubnis des Nutzers eingeholt werden. Dokumente (bzw. Dateien beliebigen Typs) sollten ab nun nur noch über das Storage Access Framework, also nach expliziter Erlaubnis des Nutzers erreichbar sein. Als letzte Maßnahme für Entwickler, die auf die Umstellung nicht schnell genug reagieren konnten, stelle Google ein requestLegacyExternalStorage-Flag für das Android Manifest bereit, das übergangsweise den Zugriff auf externen Speicher mittels File API weiter erlaubte.

3.9. Android 11 (2020)

Seit Android 11 wird Scoped Storage für alle Apps erzwungen, die als targetSdkVersion Android 11 angeben. Konkret bedeutet das, dass Apps, die auf Google Play veröffentlicht werden, zur Umstellung gezwungen werden. Um die Umstellung für Entwickler zu beschleunigen, entwickelte Google eine transparente Übersetzungsschicht: Alle Aufrufe der File-API für Dateien im öffentlichen Speicher werden an den MediaStore umgeleitet. Die Implementierung der File-API selbst wurde dazu nicht angerührt. Stattdessen findet die Übersetzung im Kernel statt.

Das requestLegacyExternalStorage-Flag im Android Manifest wird nun ignoriert, wenn die targetSdkVersion auf Android 11 gestellt wird. Stattdessen wurde eine neue MANAGE_EXTERNAL_STORAGE-Permission [6] eingeführt, die aber nur für Apps wie Datei-Manager gedacht ist, die den vollen Zugriff legitimerweise brauchen. Die Legitimität wird durch Google beim Upload in den Play Store geprüft.

4. Analyse der Sicherheit und Performance

In diesem Abschnitt werden die beiden Kernaspekte der neuen Lösung Scoped Storage analysiert, wie sie mit Android 11 vorliegt. Dazu werden auch Implementierungsaspekte und Nutzungsdaten in Produktiv-Anwendungen herangezogen.

4.1. Implementierung der transparenten Übersetzungsschicht

Für die konsequente Umsetzung der Scoped-Storage-Sandbox für alle Anwendungen fand Google herausfordernde Rahmenbedingungen vor. Das Android-Ökosystem ist bekannt dafür, dass durch die starke Fragmentierung nur ein Bruchteil der Nutzer die aktuellste Betriebssystem-Version nutzt. Entwickler haben daher wenig Motivation, neue Betriebssystem-Funktionen in ihre Anwendung zu integrieren, wenn sie zu keinem für den Nutzer ersichtlichen Funktionsgewinn der Anwendung selbst führen. Diese Gründe führten aber dazu, dass Google für die konsequente Umsetzung von Scoped Storage auch eine transparente Übersetzungsschicht zwischen altem und neuem Paradigma implementieren musste.

Aus technischer Sicht war es notwendig, diese Übersetzungsschicht im Kernel zu verankern, damit alle File-System-Syscalls abgedeckt werden können. Änderungen lediglich im Android-Framework hätten dazu geführt, dass Dateisystem-Zugriffe aus nativen Komponenten unverändert Bereiche zu manipulieren versucht hätten, die sie nicht mehr erreichen können.

In der technischen Umsetzung wurde daher entschieden, die Position des Media Store im Android-System zu verändern. Handelte es sich in früheren OS-Versionen um einen Index, der Metadaten von Dateien im Filesystem spiegelte, so wurde die Komponente für Android 11 zum Backend für das Filesystem selbst gemacht. Mittels FUSE (Filesystem in Userspace) wird das für Apps sichtbare Dateisystem vom

Media Store zur Verfügung gestellt. Es handelt sich nicht mehr um ein klassisches Dateisystem, sondern vielmehr um ein Fenster in die Datenbank des Media Store [7].

Als Teil des Boot-Prozesses wird dazu ein MediaStore-Fuse-Daemon gestartet. Das FUSE-Kernel-Modul wird so konfiguriert, dass alle Syscalls mit Dateizugriffen, die Dateien im öffentlichen Speicher betreffen, an den MediaStore-Fuse-Daemon weitergeleitet werden. Dieser entscheidet dann, welcher Datenbank-Eintrag am abgefragten Pfad verfügbar sein soll. Die vollständige Implementierung ist als Teil des Android Open Source Project öffentlich einsichtig [8].

4.2. Sicherheit

Im Vordergrund der Argumentation für Scoped Storage stand zweifelsohne die Privatsphäre der Nutzer. Solange dies möglich war, übernahmen Anwendungsentwickler unbedarft Code-Fragmente und Paradigmen von Desktop-Projekten, sodass der externe Speicher häufig ein interessantes und einfach zu erreichendes Ziel für Angreifer war, über das teils sensible Daten installierter Anwendungen erreicht werden konnten. Auch, nachdem Google erste Gegenmaßnahmen ergriffen hatte, sträubten sich viele Entwickler, die neuen optionalen Paradigmen für den Dateizugriff umzusetzen [9]. So war Google schließlich gezwungen, radikale Maßnahmen zu ergreifen.

Zur Verbesserung der Privatsphäre sollte Scoped Storage im Wesentlichen eine Sandbox für Applikationsdaten einführen, sodass Dateien nur noch geteilt werden können, die (a) zu einer definierten Kategorie von Mediendaten gehören, oder (b) vom Nutzer explizit für die Verarbeitung in einer bestimmten App vorgesehen wurden.

Die Ziele für die Umsetzung waren also [10]:

1) Unterbinden aller Zugriffe von Apps auf (nicht zur Freigabe gedachte) Daten anderer Apps
2) Apps daran zu hindern, unbedacht Daten im öffentlichen Speicher abzulegen
3) Selektiver Datenzugriff: Möglichkeit, die Privatsphäre trotz Zugriff auf Daten zu schützen. Konkret sollten trotz erlaubtem Datenzugriff die Metadaten zurückbehalten werden.

Die Entscheidung, den Media Store für umzubauen, wurde wohl aus der Formulierung dieser Ziele abgeleitet.

4.2.1. Ziel 1: Zugriff auf App-Daten unterbinden

Zur Umsetzung dieses Ziels verbietet Scoped Storage, auf den app-privaten Ordner anderer Apps zuzugreifen, selbst wenn dieser von einer App über DAC zugreifbar gemacht wird. Falls Apps Dateien aus ihrem privaten Ordner freigeben möchten, können sie einen FileProvider implementieren. Dabei handelt es sich um eine von den offiziellen Support-Libraries zur Verfügung gestellte Komponente auf Basis der ContentProvider-Infrastruktur, die den Zugriff z.B. mittels Custom Permissions absichern kann.

4.2.2. Ziel 2: Unbedachte Datenablage im öffentlichen Speicher

Dieses Ziel soll vor allem erreicht werden, indem der Media Store als einzige öffentliche Ablagemöglichkeit zur Verfügung steht, für die keine explizite Nutzererlaubnis nötig ist. Alle Dateien im Media Store müssen entsprechend ihrem MIME-Type einer Medien-Kollektion zugeordnet werden, sodass eher sichergestellt wird, dass die öffentliche Ablage vom Entwickler bewusst veranlasst wird.

Zusätzlich werden die app-spezifischen Ordner im externen Speicher nun nicht mehr für andere Apps zugänglich gemacht. Bisher hatte dieser Bereich stets den Eindruck vermittelt, wie das Äquivalent im internen Speicher von DAC geschützt zu sein, was aber nicht der Fall war.

Konzeptionell widerspricht die FUSE-Übersetzungsschicht diesem Ziel. Apps, die weiterhin die File-API nutzen können immer noch unbedarft Daten im öffentlichen Speicher ablegen, die dann von anderen Apps gelesen werden können. Denkbar ist beispielsweise ein Szenario, in dem eine App eine Java-Bibliothek einbindet, die über die File-API sensible Log-Daten im Dateisystem ablegt. Obwohl der Zugriff über die Media Store API umgeleitet wird, sind die Daten für andere Apps einsehbar.

4.2.3. Ziel 3: Selektiver Datenzugriff

Der Media Store übernimmt hier eine Filterfunktion, um Mediendaten vor der Weitergabe an App von sensiblen Metadaten zu befreien. Dazu Wird auch in der FUSE-Übersetzungsschicht die Möglichkeit implementiert, Datei-Bereiche zu hinterlegen, die nicht an Apps weitergereicht werden. Konkret genutzt wird diese Möglichkeit etwa dazu, Bild-Dateien vor der Auslieferung von EXIF-Metadaten zu befreien, die Anwendungen Aufschluss über den Aufenthaltsort des Nutzers geben könnten.

4.2.4. Effektivität der umgesetzten Lösung

Die umgesetzte Lösung ist in der Lage, die zum Ziel gesetzten Aufgaben großteils zu erfüllen. Unterbunden werden etwa Schwachstellen, die dadurch entstanden waren, dass der app-spezifische Speicher von Apps auch für andere Anwendungen zugreifbar war. Diese Schwachstellen waren zuvor weit verbreitet und betrafen zum Teil auch namhafte Anwendungen. 2021 zum Beispiel war eine Sicherheitslücke im bekannten WhatsApp-Messenger gefunden worden, durch die Angreifer TLS-Verbindungen abhören konnten [11]. Ursache des Problems war, dass Session-Keys im app-spezifischen Ordner im externen Speicher abgelegt worden waren. Das Problem konnte auf Android-Systemen mit Android 10 oder niedriger ausgenutzt werden, nicht aber unter Android 11, wo Scoped Storage für alle Anwendungen erzwungen wird.

In der Literatur sind bisher nur wenige Untersuchungen zur allgemeinen Effektivität von Scoped Storage zu finden. Heid et al. [9] fanden heraus, dass 47% der Anwendungen im Google Play Store mit Target-SDK-Version 29 (Android 10) Scoped Storage mittels requestLegacyExternalStorage-Flag umgehen. Insgesamt kommen sie zu dem Schluss, dass das Erzwingen von Scoped Storage die Sicherheit und Privatsphäre der Nutzer erhöht. Zwar wurden keine Apps gefunden, die sensible Daten mittels FUSE-Übersetzungsschicht weiterhin im gemeinsamen Speicher ablegen, zur Analyse wurde allerdings nur ein relativ simpler Ansatz gewählt, in dem Apps zufälligem User-Input ausgesetzt waren.

4.3. Performance

Google selbst beschreibt die Implementierung von Scoped Storage als eine Abwägung zwischen Privatsphäre und Performance [7]. Um den Performance-Overhead für die Nutzer ihrer Anwendungen möglichst gering zu halten, sind Entwickler dazu angehalten, die MediaStore-API zu nutzen. Obwohl besonderer Wert auf eine effiziente FUSE-Implementierung gelegt wurde, ist hier mit dem größten Overhead zu rechnen.

Beim Studium der Implementierung fällt auf, dass für jeden File-System-Zugriff über die File-API mehrere rechenintensive Operationen ausgeführt werden müssen. Aus der App heraus wird die Ausführung zunächst wie üblich per Syscall an den Kernel übergeben. Von hier wird über das FUSE-Kernel-Modul der FUSE Daemon aufgerufen. Dieser wiederum greift mittels Java Native Interface (JNI) auf den eigentlichen Media Store zu, von wo erst eine SQLite-Datenbank und das zugrundeliegende Dateisystem befragt

werden. Aus der Summe dieses komplexen Zusammenspiels verschiedener Komponenten ergibt sich ein nicht unerheblicher Performance-Overhead, wenn die FUSE-Übersetzungsschicht genutzt wird.

Abbildung 1 Dateizugriff über die FUSE-Übersetzungsschicht in Android 11

Abbildung 2 Vereinfachter Prozess für den Dateizugriff vor Android 10

Google gibt an, dass für die Implementierung der FUSE-Übersetzungsschicht einige Performance-Optimierungen realisiert wurden. Es werden etwa manche Pfade, für die keine Filterung notwendig ist, mittels eBPF direkt vom FUSE-Kernel-Modul an das darunterliegende Dateisystem weitergeleitet. Mehrere Caches an verschiedenen Stellen sollen ebenfalls die Performance verbessern.

Der Media Store wird außerdem seit Android 11 als eigenständiges System-Modul entwickelt, das unabhängig vom Android-System über den Play Store aktualisiert werden kann [7]. So wurde etwa im Oktober 2020 per Komponenten-Update die Performance für Apps verbessert, die über die MANAGE_EXTERNAL_STORAGE-Permission verfügten.

Selbst wenn die FUSE-Übersetzungsschicht nicht genutzt wird, sondern mittels ContentProvider-API direkt auf den Media Store zugegriffen wird, ist mit Performance-Einbußen im Vergleich zu direktem Dateizugriff zu rechnen. Noch immer wird dann der Datenzugriff über IPC (Inter Process Communication – in diesem Fall genauer das Android-spezifische Binder-Subsystem) an den MediaStore geleitet, der dann den Datenzugriff im zugrundeliegenden File-System regelt. Für jeden Aufruf wird daher im Vergleich zum direkten File-System-Zugriff eine zusätzliche IPC-Transaktion und der in Java implementierte Media Store aufgerufen. Um den entstehenden Overhead zu minimieren, empfiehlt Google, Bulk- bzw. Batch-Operationen [12] am Media Store auszuführen.

Der Vollständigkeit halber wird hier noch erwähnt, dass Zugriffe auf Dateien im internen app-privaten Speicher von keinen Performance-Einbußen betroffen sind, da sie nicht über FUSE gemountet werden [13]. Zugriffe auf Dateien im externen app-privaten Speicher sind von den Einbußen nur mäßig betroffen, da sie mittels eBPF bypassed werden. Hierbei wird vom FUSE-Daemon ein eBPF-Programm beim FUSE-Kernel-Modul hinterlegt, das dann direkt im Kernel ausgeführt wird, um zu entscheiden, ob ein bestimmter Zugriff an den Daemon weitergeleitet werden soll. Die Zugriffe auf app-privaten externen Speicher werden so schon vom Kernel an das darunterliegende File-System weitergeleitet, ohne den Umweg über den Daemon nehmen zu müssen.

4.4. Performance-Evaluierung

Um einen quantitativen Eindruck der Performance-Auswirkungen von Scoped Storage und er FUSE-Übersetzungsschicht zu erhalten, wurde im Rahmen dieses Projekts ein Benchmark realisiert. Dabei wurden für Direkten Filesystem-Zugriff, MediaStore-Zugriff und FUSE-Zugriff Laufzeiten gemessen. Für jeden Test wurden je 1000 Dateien zu 1 MB erstellt, sequentielle Daten geschrieben, gelesen und gelöscht.

Für jedes Experiment wurde der Mittelwert aus 10 Samples ermittelt. Es wurde ein Google Pixel 3 mit Android 11 verwendet.

Wie in der untenstehenden Grafik ersichtlich zeigte sich, dass wie erwartet das Schreiben per direktem Filesystem-Zugriff am schnellsten war. Die MediaStore-Schnittstelle zeigte sich gut 5x langsamer (502%), war aber dennoch schneller als FUSE-Zugriff. Letzterer war fast 6x langsamer als der direkte Zugriff (583 %). Beim Lesezugriff waren die Unterschiede sogar noch deutlich größer (MediaStore: 658% overhead, FUSE: 1053%). Die größten Unterschiede zeigten sich beim Löschen der Dateien: MediaStore war 36x (35505%) langsamer als direkter Zugriff, FUSE 33x (32715%). Hier schnitt die MediaStore-API interessanterweise knapp schlechter ab als FUSE.

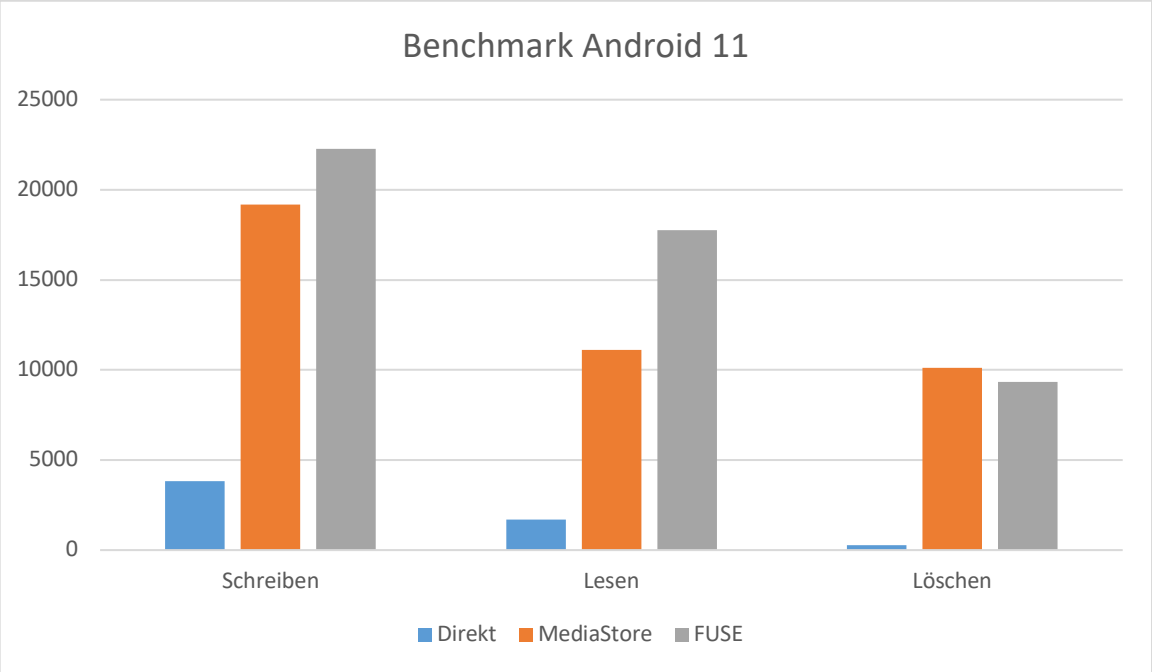

Abbildung 3 Messergebnisse unseres Dateizugriff-Benchmarks. Messungen in Millisekunden.

In der Analyse unserer Messergebnisse zeigen sich deutliche Unterschiede zu Googles eigenen Performance-Angaben [7]:

- Der Lese-Overhead ist größer als der Schreib-Overhead
- Es kann keineswegs von vergleichbarer Leseperformance zwischen FUSE und MediaStore gesprochen werden (160% overhead).

Die Angabe, dass Zugriff auf das externe app-spezifische Verzeichnis im externen Speicher ähnlich schnell wie direkter Zugriff (app-privater Ordner im internen Speicher) sei, konnten wir experimentell bestätigen.

Angesichts der ermittelten Overheads sowohl der MediaStore-API als auch der FUSE-Übersetzungsschicht muss damit gerechnet werden, dass das Nutzererlebnis für Anwendungen unter Android 11 durch Scoped Storage deutlich getrübt wird. Hier wäre es wünschenswert gewesen, wenn Google die Implementierung des MediaStore effizienter gestaltet hätte, etwa durch eine rein native Implementierung.

4.5. Prävalenz der File-API

Um die Prävalenz von Anwendungen, die die FUSE-Übersetzungsschicht nutzen, feststellen zu können, wurde ein Tool gebaut, das den Dalvik-Bytecode von kompilierten Android-Anwendungen auf

Verwendungen der File-API durchsucht. Mittels dieses Tools wurden dann die populärsten 100 Android-Anwendungen aus 33 Kategorien in Googles Play Store untersucht (einige Anwendungen konnten nicht heruntergeladen werden, so dass in Summe 2701 APK-Files untersucht wurden).

Unsere Analyse zeigte, dass 363 der untersuchten 2701 Anwendungen (13,4%) weiterhin ausschließlich Dateizugriffe mittels der Legacy-File-API implementieren und dadurch von den Performance-Einbußen der FUSE-Übersetzungsschicht betroffen sind.

Angesichts des weit verbreiteten Nutzens der FUSE-Übersetzungsschicht kann festgehalten werden, dass ein erheblicher Teil von Android-Anwendungen zwar von der verbesserten Privatsphäre von Scoped Storage profitiert, allerdings auch deutliche Performanceeinbußen zeigt.

4.5.1. Ideen für weitere Projekte

Auf Basis der ausführlichen Implementationsanalyse dieses Berichts könnte in einem weiteren Schritt untersucht werden, inwieweit sich das Modell Scoped Storage auch auf ältere Android-Versionen übertragen lässt. Denkbar wäre etwa, mittels Runtime Hooking [14] Aufrufe der Java File-API auf MediaStore-Zugriffe umzuleiten. Ergänzend könnten bei einer systemweiten Lösung auch die Zugriffsbeschränkungen für die app-spezifischen Ordner auf externem Speicher umgesetzt werden. Eine Herausforderung am Weg zur Umsetzung dieser Ideen ist die Notwendigkeit, auch Dateizugriffe aus nativem Code umzuleiten. Hierzu könnten etwa Lösungen zum Einsatz kommen, die nativen Code patchen [15] oder die Global-Offset-Table-Einträge für libc-Syscall-Wrapper manipulieren [16].

Die Implementierung rein im Userspace hätte den Vorteil, dass weniger aufwendige Kontext-Wechsel als bei der FUSE-Übersetzungsschicht nötig wären. Die Implementierung könnte also performanter verwirklicht werden, sodass sie auch für neuere Android-Versionen relevant wäre.

5. Zusammenfassung

In diesem Projekt wurde das Scoped-Storage-Modell des Android-Betriebssystems aus verschiedenen Gesichtspunkten beleuchtet. Es wurde ein Überblick über das Speichermodell der Plattform im Laufe der Zeit präsentiert und die Aspekte Sicherheit und Performance ausgiebig diskutiert. Es konnte gezeigt werden, dass Scoped Storage zwar deutliche Vorteile in Bezug auf die Privatsphäre der Nutzer bietet, allerdings auch spürbare Auswirkungen auf die Performance von Dateizugriffen hat.

Referenzen

[1] Google, „Android Developers," 2009. [Online]. Available: https://web.archive.org/web/20090627025740/http://developer.android.com/guide/topics/data/data-storage.html#files.

[2] Google, „Android Developer Documentation: Context," 2010. [Online]. Available: https://developer.android.com/reference/android/content/Context#getExternalFilesDir(java.lang.String).

[3] Google, „Android API Documentation: MediaStore.Files," 2011. [Online]. Available: https://developer.android.com/reference/android/provider/MediaStore.Files.

[4] Google, „Android 7.0 for Developers," 2016. [Online]. Available: https://developer.android.com/about/versions/nougat/android-7.0?hl=en#scoped_directory_access.

[5] Google, „Direct document access," 2017. [Online]. Available: https://developer.android.com/about/versions/oreo/android-8.0#dda.

[6] Google, „Android Developers API Reference: Manifest.permission," 2020. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission#MANAGE_EXTERNAL_STORAGE.

[7] Google, „Android Open Source Project: Scoped Storage," 2021. [Online]. Available: https://source.android.com/docs/core/storage/scoped#mediaprovider-updates.

[8] Google, „Android Code Search: FuseDaemon," 2019. [Online]. Available: https://cs.android.com/android/platform/superproject/+/master:packages/providers/MediaProvider/jni/FuseDaemon.cpp.

[9] K. Heid, T. Tefke, J. Heider und R. C. Staudemeyer, „Android Data Storage Locations and What App Developers Do with It from a Security and Privacy Perspective," in *Proceedings of the 8th International Conference on Information Systems Security and Privacy* , 2022.

[10] Google, „Android Developer Documentation: Data and file storage overview," 2019. [Online]. Available: https://developer.android.com/training/data-storage#scoped-storage.

[11] C. Karamitas, „WhatsApp exposure of TLS 1.2 cryptographic material to third party apps," 2021. [Online]. Available: https://census-labs.com/news/2021/04/14/whatsapp-exposure-of-cryptographic-material-to-third-party-apps/. [Zugriff am 2022].

[12] Google, „Android API Documentation: ContentProvider," [Online]. Available: https://developer.android.com/reference/android/content/ContentProvider#applyBatch(java.util.ArrayList%3Candroid.content.ContentProviderOperation%3E).

[13] Google, „Android Open Source Project: Scoped Storage - Mitigating FUSE," [Online]. Available: https://source.android.com/docs/core/storage/scoped#mitigating-fuse.

[14] F. Draschbacher, „On-Device Patching-Framework für Android-Anwendungen," 2021. [Online]. Available: https://technology.a-sit.at/on-device-patching-framework-fuer-android-anwendungen/.

[15] Y. Zhou, K. Patel, L. Wu, Z. Wang und X. Jiang, „Hybrid User-level Sandboxing of Third-party Android Apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015.

[16] M. Backes, S. Bugiel, C. Hammer, O. Schranz und P. v. Styp-Rekowsky, „Boxify: Full-fledged App Sandboxing for Stock Android," in *24th USENIX Security Symposium* , 2015.