A=SIT Technology
E-Government Innovationszentrum

21. Dezember 2021 Studie · Mobile Sicherheit · 16 Min. Lesezeit

On-Device Patching-Framework für Android-Anwendungen

Archiviert. Dieser Inhalt wird nicht mehr aktualisiert. Er bleibt zur Information verfügbar.

Stand 2021-12-13
Quelle https://technology2.a-sit.at/veroeffentlichungen/on-device-patching-framework-fuer-android-anwendungen/

Das Patchen von bestehenden Android-Applikationen hat ein breites Anwendungsspektrum, das von der Behebung von Sicherheitslücken über die Instrumentierung zum Zwecke der Malware-Analyse bis hin zur Funktionserweiterung von Software reicht.

Bestehende Lösungen beschränken sich allerdings auf das reine Patchen des Applikationscodes, der nur einen Bestandteil von Android-Anwendungen darstellt. Zusätzlich können hier Patch-Entwickler üblicherweise nur konkrete Anwendungen patchen, oder zumindest nur bestimmte Klassen. Häufig ist es aber auch notwendig, Klassen anhand ihrer Vererbungshierarchie zu bearbeiten, oder neben Änderungen am Applikationscode auch Ressourcen einzufügen oder das Android-Manifest zu modifizieren.

Im Rahmen dieses Projektes wurde ein voll funktionsfähiges Patching-Framework geschaffen, bestehend aus einem Entwicklungs-Kit für die einfache Erstellung applikations-agnostischer Patches, sowie einer kompletten Patching-Pipeline, die sich aufgrund ihrer hohen Effizienz auch zur unkomplizierten Integration in mobile Anwendungen eignet.

1. Einleitung

Der Aufstieg des Mobilbetriebssystems Android während der letzten eineinhalb Jahrzehnte wurde begleitet vom Entstehen eines immer aktiveren Forschungsfeldes, das sich speziell der Erforschung der Sicherheit verschiedener System-Komponenten aber auch des Applikations-Ökosystems widmet. Im Rahmen dieser Tätigkeiten versuchen Wissenschaftler etwa, Sicherheitslücken in Programmen von Drittanbietern aufzudecken, Einblicke in die Funktionsweise potentieller Schadsoftware zu erlangen, oder strukturelle Schwachstellen in der Sicherheitsarchitektur des System-Frameworks zu identifizieren und gezielt zu beheben. Ein essentielles Werkzeug zur Umsetzung dieser Vorhaben ist die Technik des Applikations-Patchings, mit deren Hilfe bestehende Anwendungen modifiziert werden können, ohne Zugriff auf den ursprünglichen Programmcode zu haben.

Obwohl aufgrund des umfassenden Bedarfs auch von Seiten der Wissenschafts-Community schon einige Software-Pakete vorgestellt wurden, die das Patchen von kompilierten Android-Anwendungen unterstützen, leiden sie bisher doch alle unter mindestens einer der folgenden erheblichen Einschränkungen L1-L5.

L1: Coverage Viele aktuell erhältliche Tools konzentrieren sich ausschließlich auf das Modifizieren des Kontrollflusses des Programmcodes der Ziel-Anwendungen. Zwar stellt der Programmcode den zentralen Kern zur Implementierung der Funktionalität eines Programms dar, doch sind auch andere Elemente entscheidend für die Nutzererfahrung. Das Android-Manifest beispielsweise definiert die Interaktionspunkte einer Anwendung mit dem System. Kann das Manifest nicht verändert werden, können keine neuen Activities, ContentProvider, BroadcastReceiver oder Services registriert werden.

L2: Skalierbarkeit Andere Tools arbeiten nicht applikations-agnostisch, erlauben also nur das Anpassen einer konkreten vorliegenden App-Version. In vielen Fällen ist es aber wünschenswert, Modifikationen als Change-Sets generisch zu formulieren, um dieselben Änderungen beliebig skalierbar auf mehrere Applikationen anwenden zu können.

L3: Praktikabilität Eine weitere häufige Einschränkung ist der Bedarf von Root-Berechtigungen einiger Lösungen. Zwar stellt diese Anforderung in geschlossenen Forschungs-Szenarien kein großes Problem dar, soll aber ein gepatchtes Programm etwa für breitflächigere Tests auch an weniger erfahrene Nutzer ausgerollt werden, so ergeben sich erhebliche Schwierigkeiten. Root-Rechte sind nicht nur relativ schwierig zu erlangen, sie können durch unbedachte Verwendung auch schnell zu schwerwiegenden Fehlern wie ungewollte Datenlöschung bis hin zur versehentlichen Deaktivierung wichtiger Sicherheits-Mechanismen führen.

L4: Effektivität oder Effizienz Zusätzlich müssen sich Anwender und Forscher bei der Wahl eines Patching-Systems bisher zwischen hoher Effektivität (insbesondere Kompatibilität mit verschiedenen Android-Versionen und -Geräten) und hoher Effizienz (schnelles Patch-Deployment) entscheiden. Ändern sich die Anforderungen, steht keine Möglichkeit zur Verfügung, ein bestehendes Patch-Projekt ohne radikale Überarbeitung zu einer anderen Technologie zu übertragen.

L5: System-Integration Als letzte bekannte Einschränkung bestehender Lösungen ist die System-Integration zu erwähnen. Läuft eine modifizierte Anwendung innerhalb einer Container-Anwendung, kann Erstere nicht wie bei einer üblichen Installation im System registriert werden. Weil aus Sicht des Systems jede Interaktion mit der Ziel-App über die Container-Anwendung läuft, muss diese statisch alle Berechtigungen anfordern, die von einer Ziel-App jemals benötigt werden könnten. Die Trusted Compute Base (TCB) wird also signifikant vergrößert und dadurch deutlich anfälliger für Angriffe.

Framework	L1	L2	L3	L4	L5
Apktool [1]		x		x	
Xposed [2]	x		x	x	
DexPatcher [3]		x		x	
VirtualXposed [4]	x			x	x
Cydia Substrate [5]	x		x	x	
Frida [6]	x			x	
ARTDroid [7]	x		x	x	
Reptor [8]	x			x	
RetroSkeleton [9]	x		x	x	

Tabelle 1 *: Limitierungen bestehender Patching-Frameworks*

Wie aus *Tabelle 1* ersichtlich, sind alle bisher veröffentlichten Patching-Frameworks von mindestens zwei Limitierungen betroffen. Im Gegensatz dazu ist das im Rahmen dieses Projekts entwickelte Framework von keiner der Einschränkungen L1-L5 betroffen.

2. Hintergrund

2.1. APK-Format

Android-Applikationen werden in Java oder kompatiblen Sprachen verfasst und während des Build-Vorganges in ein spezielles Dateiformat namens APK überführt. Hierbei handelt es sich um ein leicht angepasstes ZIP-Archiv, das eine vordefinierte Dateistruktur enthält:

- AndroidManifest.xml
- resources.arsc
- classes.dex (Optional: classes-1.dex, classes-2.dex, ...)
- res/
- assets/
- libs/

Die für dieses Projekt entscheidenden Bestandteile werde im Folgenden näher erläutert.

2.2. AndroidManifest.xml

Die Manifest-Datei wird schon vom Applikationsentwickler als XML-Datei angelegt, in der die Interaktionspunkte der App mit dem Betriebssystem vermerkt werden. Beispiele sind hier etwa `Activities` (UI-Fenster), oder `Services`, `ContentProvider` und `BroadcastReceiver`, die es der App jeweils erlauben, auf Systemereignisse zu reagieren oder anderen Prozessen Daten zur Verfügung zu stellen. Außerdem muss eine Liste der benötigten (oder zur Laufzeit angefragten) Berechtigungen hinterlegt werden. Als Teil des Build-Prozesses wird das Manifest in ein proprietäres Binary-XML-Format übersetzt und so im APK-File gespeichert.

2.3. ARSC-Format

Android-Apps können eine Vielzahl verschiedener Ressource-Typen enthalten, zum Beispiel Layout-XMLs (zur Beschreibung der Struktur der UI), Vektor- oder Rastergrafiken für Icons oder Theming, String-Tabellen in verschiedenen Sprachen, oder Konfigurations-Dateien. Alle diese Ressourcen werden durch einen eindeutigen Integer identifiziert, und müssen zusätzlich mit einem Namen versehen werden. Das Android-SDK nutzt diese Namen dann zur Generierung einer Java-Klasse mit einem als `final static` markierten Integer-Feld für jede Ressource, das vom Applikationscode anstelle des hardcoded Integer-Identifikators referenziert werden kann.

Um Speicherplatz am Android-Gerät zu sparen, werden viele Ressourcen während der App-Kompilierung komprimiert. Als Teil dieses Prozesses wird ein Index-Table für alle Ressourcen angelegt. Dieser Index (und auch kompilierte String-Ressourcen) werden in einer speziellen Datenstruktur in der resources.arsc-Datei gespeichert. Die Datei ist in verschachtelten Chunks strukturiert. An der Wurzel der Struktur sitzt dabei der Table Chunk, der ein oder mehrere Package Chunks enthält. Letztere enthalten wiederum TypeSpec Chunks (einen für jeden Ressource-Typ), sowie Type Chunks, die einen Ressource-Wert für eine bestimmte Gerätekonfiguration enthalten.

Die Integer-Identifier, die für das Referenzieren der Ressourcen verwendet werden, kodieren jeweils einen Pfad in die beschriebene verschachtelte Struktur. Zum Beispiel kann die Ressource-ID `0x7f0b001e` interpretiert werden als der 30. (`0x001e`) Ressource-Eintrag des 11. (`0x0b`) Type-Chunks im Package `0x7f` . Erwähnt werden muss hier noch, dass die Android-Build-Tools standardmäßig alle Ressourcen einer App (auch die von Library-Dependencies) in einen einzigen Package Chunk mit der ID `0x7f` speichern.

2.4. DEX-Format

Ein oder mehrere DEX-Files innerhalb der APK-Dateistruktur enthalten den kompilierten Programmcode im speziellen Dalvik-Bytecode-Format. Die Dateien werden während des Kompiliervorgangs aus dem von den App-Entwicklern geschriebenen Java-Code (oder Code in kompatiblen Sprachen) erzeugt. Soll eine Anwendung am Android-Gerät ausgeführt werden, wird dazu der Dalvik-Bytecode von der ART-Runtime zuerst interpretiert und später Stück für Stück in nativen Code übersetzt (Just-In-Time-Compilation sowie Ahead-Of-Time-Compilation).

3. System-Aufbau

Als Alternative zu den wie in der Einleitung erwähnten von erheblichen Einschränkungen betroffenen bestehenden Patching-Systeme für Android-Anwendungen wurde im Rahmen dieses Projekts ein neues Framework geschaffen, das als solide Basis für weitere Forschungen dienen soll. Patch-Projekte können bequem direkt in der schon aus der Applikationsentwicklung bekannten Android-Studio-IDE erstellt und später auf beliebige Ziel-Pakete angewandt werden. Dabei sind die folgenden Kernmerkmale zu erwähnen:

3.1. Holistisches Patchen

Patch-Projekte können nicht nur Änderungen am Kontrollfluss einer Ziel-Anwendung formulieren, sondern auch das Android-Manifest modifizieren, sowie neue Ressourcen, native Bibliotheken oder Assets einfügen. Für die Spezifikation all dieser Änderungen stehen spezielle Dateiformate zur Verfügung, die jeweils eng an die entsprechenden Original-Dateien im ursprünglichen App-Projekt angelehnt sind. Damit wird sichergestellt, dass sich Anwendungsentwickler besonders schnell in der Patch-Entwicklung zurechtfinden.

3.2. Zweistufiges Verfahren

Um die Entwicklung applikations-agnostischer Patches zu erlauben und diese dennoch performant anwenden zu können, wurde ein zweistufiges Patch-Verfahren umgesetzt. Im ersten Schritt („Patch Build“) wird aus dem Patch-Projekt ein Patch-Paket erzeugt. Hierzu wird von der bestehenden Android-Toolchain Gebrauch gemacht, die um einen zusätzlichen Annotation Processor erweitert wurde. Das Patch-Paket folgt in Format und Aufbau in weiten Teilen dem APK-Format. Als zweiter Schritt („Patch Deployment“) wird dann eine Patch-Deployment-Pipeline aufgerufen, die die im Patch formulierten Änderungen auf das APK-File einer Ziel-App anwendet. **Abbildung 1** zeigt eine schematische Darstellung des zweistufigen Verfahrens und aller involvierten Komponenten.

3.3. Austauschbare Rewriting-Backends

Viele bestehende Patching-Systeme bauen für die Modifikation des Kontrollflusses gepatchter Apps während des Patch-Deployments entweder auf das Anpassen des im APK enthaltenen Dalvik-Bytecodes („statisches Rewriting“) oder verändern zur Laufzeit die internen Datenstrukturen der ART-Runtime („dynamisches Rewriting“). Während der erste Ansatz verlässlich funktioniert, ist er aufgrund der Komplexität des DEX-Formats sehr rechenintensiv. Im Gegensatz dazu ist der zweite Ansatz deutlich performanter, muss allerdings häufig adaptiert werden, weil sich die interne Funktionsweise der ART-Runtime mit jeder neuen Android-Version ändert. Als Lösung für dieses Problem integriert unser Framework zwei verschiedene Rewriting-Backends, die beide beschriebenen Ansätze implementieren. Vor dem Patch-Deployment kann dann auf Basis der Zielumgebung entschieden werden, welches Backend verwendet wird. So kann die Funktionalität auch auf neuen Android-Versionen gewährleistet werden, für die das dynamische Rewriting noch nicht adaptiert wurde. Die konkrete Rewriting-Technologie wird durch den Annotation Processor und die Deployment-Pipeline vom Patch-Code abstrahiert, so dass sie für Patch-Entwickler keine Rolle spielt.

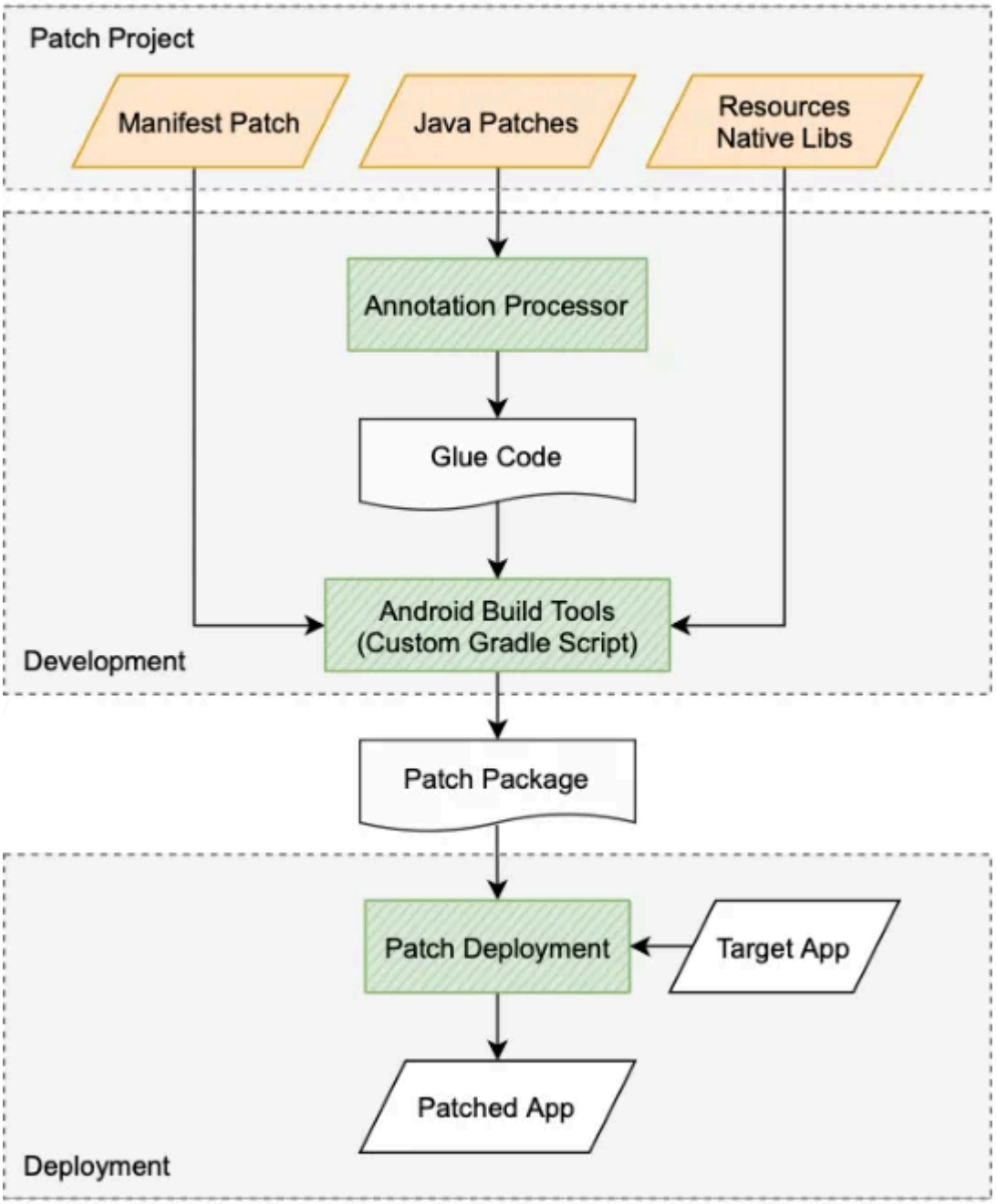
Komponenten

4. Patch-Entwicklung

Da unser Patching-Framework auf die bestehenden Android-Build-Tools aufbaut, können Patches direkt in der offiziellen Android-Studio-Entwicklungsumgebung gestaltet werden. Das Projekt-Format wurde dazu sehr stark an jenes normaler Applikations-Projekte angelehnt, einzig der Gradle-Build-Prozess muss mittels Modifikation von `build.gradle` leicht angepasst werden.

4.1. Java-Patches

Das Grundprinzip des unterstützen Patchens des Programmflusses einer Ziel-Anwendung ist das Ersetzen von Methoden-Implementierungen. Anstelle einer von der Ziel-Anwendung aufgerufenen Methode wird dann eine vom Patch-Entwickler spezifizierte Methode aufgerufen, die ihrerseits bei



tische Darstellung des zweistufigen Patch-Verfahrens

Bedarf die ursprüngliche Implementierung aufrufen kann. Diese Ersatzmethoden werden dazu im Patch-Projekt als statische Methoden in einem beliebigen Java-File hinterlegt. Mittels einer Java-Annotation kann dann die zu ersetzende Methode spezifiziert werden. **Listing 1** zeigt einen einfachen Beispiel-Patch. Hier werden zwei Instanz-Methoden der Klasse `android.content.Intent` gepatcht.

```
@PatchClass( "android.content.Intent" )
public static class IntentPatch {
  @PatchInstanceMethod
  public static Intent setClassName(Intent thiz, String packageName,
                      String className) {
     if (packageName.equals( getUnpatchedPackageName ())) {
packageName = getPatchedPackageName ();

     return OriginalMethods.android_content_Intent. setClassName (thiz,
packageName, className); @PatchInstanceMethod public static Intent
setComponent(Intent thiz, ComponentName c) { if (c ≠ null &&
c.getPackageName().equals( getUnpatchedPackageName ())) { c = new ComponentName(
getPatchedPackageName (), c.getClassName()); return
OriginalMethods.android_content_Intent. setComponent (thiz, c);
```

***Listing 1** : Ein Beispiel-Patch für den Programmfluss einer Ziel-App* Analog zur `PatchInstanceMethod` -Annotation stehen auch Annotations für Konstruktoren und Klassenmethoden zur Verfügung. Wie in **Listing 1** ersichtlich, unterscheidet sich für Instanz-Methoden die Signatur der Patch-Methode von der der Ziel-Methode: Weil Patch-Methoden immer statisch sein müssen, wird hier das Instanz-Objekt („ `this` “) als erstes Argument explizit übergeben. Die Original-Methoden können über die `OriginalMethods` -Klasse aufgerufen werden, die vom Annotation Processor erzeugt wird. Details dazu werden in 4.4 beschrieben.

4.2. Manifest-Patches

Für die Modifikation des Android-Manifests werden Patch-Entwicklern umfangreiche Möglichkeiten in Form eines eigens entwickelten XML-Patch-Formats auf Basis von RFC5261 [10] angeboten. Das Format spezifiziert Änderungen am Manifest in Form eines XML-Files, dessen Elemente je eine Änderungs-Operation beschreiben. Diese können Elemente oder Attribute hinzufügen, entfernen oder ersetzen. Zur Adressierung eines zur Änderung vorgesehenen Elements wird jede Änderungs-Operation mit einem XPath-Selektor ausgezeichnet. Dieser erlaubt die Elementauswahl anhand einer Vielzahl verschiedener Eigenschaften, etwa Attributwerte oder Verwandtschaftsverhältnisse zu anderen Elementen. Ein Beispiel-Patch ist in **Listing 2** dargestellt.

Im Beispiel sichtbar sind auch die Platzhalter (im Format `$${x}`), die es erlauben, auch neu eingefügte Werte mit Inhalten aus anderen Teilen des Ziel-Manifests zu füllen. Das `xpath( )`Prädikat erlaubt dazu die Auswahl eines referenzierten Elements, `globalize()` dient dazu, aus einem relativen Klassenpfad und dem Paketnamen einen absoluten Klassenpfad zu erzeugen, und `appendeach()` kann verwendet werden, um an jedes Element eines komma-separierten referenzierten String einen Suffix anzuhängen.

Bemerkenswert ist hier noch, dass das Patch-Manifest selbst ein gültiges App-Manifest ist, was den Entwicklungsprozess vereinfacht. Die Manifest-Patches werden einfach im Android-Manifest des Patch-Projektes gesammelt.

<?xml version=“1.0” encoding=“UTF-8”?> <manifest xmlns:android=“http://schemas.android.com/apk/res/android" xmlns:patch=“http://schemas.android.com/apk/res–auto” package=“com.apdk.sample.patch”>

<patch:add sel="manifest"> <meta-data android:name="patched" android:value="true" /> </patch:add> <patch:replace sel="manifest/@package"> $${xpath(".")}.patched <patch:replace sel="manifest/application/activity/@name"> $${globalize(xpath("/manifest/@package"), xpath("."))} <patch:replace sel="manifest/application/provider/@authorities"> $${appendeach(xpath("."), ".patched")} <patch:remove sel="manifest/application/@testOnly" /> </manifest> ***Listing 2** : Ein Beispielpatch für das Android-Manifest*

4.3. Injektion von Ressourcen, Nativen Bibliotheken oder Assets

Ressourcen, Native Bibliotheken und Assets (Ressource-Files, die während des Kompilierens nicht verändert werden) können in Patch-Projekte eingebunden und referenziert werden, wie dies auch bei normalen App-Projekten geschehen würde.

4.4. Patch-Kompilation

Während des Kompilierens eines Patch-Projektes erzeugt der Annotation Processor Glue-Code, der die Spalten zwischen Patch-Code und den verschiedenen Rewriting-Backends überbrückt. In diesem Schritt wird auch die `OriginalMethods` -Klasse erzeugt. Auch ein Teil des unter 5.1 beschriebenen Patch-Inference-Verfahrens wird hier ausgeführt. Danach werden Java-Patches und Glue-Code in Dalvik-Bytecode übersetzt und Manifest-Patch sowie Ressourcen in das proprietäre binäre XML-Format des Android-Systems überführt. Das Resultat ist ein Patch-Package, ein ZIP-Archiv, das dem Aufbau einer APK-Datei folgt, jedoch selbst weder installiert noch ausgeführt werden kann.

5. Patch-Deployment

Für das Anwenden eines Patch-Paketes auf eine APK-Datei wurde eine eigene Patch-Deployment-Pipeline geschaffen. Sie beherrscht nicht nur die entsprechende Modifikation der DEX-, ARSC- und Manifest-Files, sondern implementiert auch das notwendige Alignen des komprimierten APK-Archivs sowie das Signieren mit verschiedenen Versionen des APK-Signatur-Schemas. Die gesamte Pipeline wurde als performante Java-Bibliothek implementiert, sodass sie plattformunabhängig betrieben und auch direkt in Android-Anwendungen integriert werden kann.

5.1. Java-Patches

Das Deployment der Java-Patches unterscheidet sich zwischen den beiden Rewriting-Backends. Gemeinsam ist den beiden Varianten jedoch, dass das DEX-File aus dem Patch-Package in das gepatchte APK kopiert wird. Außerdem wird in beiden Varianten Patch-Inference betrieben, die Patches auf alle von der adressierten Ziel-Klasse abgeleiteten, von der App definierten Klassen überträgt. Dieser Schritt ist notwendig, um Anwendungsszenarien abzudecken, in denen die konkrete Ziel-Implementierung dem Patch-Entwickler nicht bekannt ist, bzw. im Patch-Code nicht direkt adressiert werden kann, um die Eigenschaft der App-Agnostizität nicht zu verletzen.

5.1.1. Statisches Rewriting

Beim statischen Rewriting werden für jede zu patchende Methode alle Aufrufe im DEX-Bytecode auf die Ersatz-Methode umgeschrieben. Zusätzlich muss für Patches von Instanz-Methoden der Op-Code korrigiert werden, da Patch-Methoden immer statisch sein und daher mit einem anderen Op-Code

aufgerufen werden müssen. Ein zusätzlicher Spezialfall ergibt sich bei Konstruktor-Patches: Die ART-Runtime erlaubt uninitialisierte Objekte nur als Parameter an Konstruktor-Aufrufe zu übergeben. Um diese Einschränkung zu umgehen, wird der Aufruf des Konstruktor-Patches als zusätzlicher Methodenaufruf direkt nach dem Konstruktor-Aufruf realisiert. Dazu müssen allerdings alle anderen Bytecode-Abschnitte der Methode adaptiert werden, die für die Referenzierung einer späteren Instruktion deren Offset in der Methode verwenden. Konkret betrifft dies Try-Blöcke, Switch-Blöcke und Exception-Handler.

Zum Parsen und Schreiben des DEX-Formats wird hier die unter einer Open-Source-Lizenz stehende dexlib2-Bibliothek [1] eingebunden.

5.1.2. Dynamisches Rewriting

Beim dynamischen Rewriting werden zur Laufzeit Änderungen an den internen Datenstrukturen der ART-Runtime vorgenommen. Dabei können Patches on-demand angewandt werden, sobald die betroffene Klasse geladen wird. Für Details zu diesem Rewriting-Backend wird hier auf den früheren Bericht „Android Application Patching durch Runtime-Manipulation“ verwiesen.

5.2. Ressource-Patches

Für das Anwenden eines Patches auf die Ressourcen einer Ziel-App wird die ARSC-Struktur aus dem Patch-Paket in die ARSC-Struktur der Ziel-App migriert. Dazu müssen nicht nur die TypeSpec-Chunks und die Type-Chunks kopiert werden, sondern auch die von ihnen referenzierten Strings in die jeweiligen StringPool-Chunks übernommen werden. Eine erhebliche Herausforderung ergibt sich hier allerdings durch die Ressource-Ids: Da der Patch als separates Projekt ohne Kenntnis der Ziel-Apps entwickelt wird, treten hier ohne besondere Vorkehrungen Überschneidungen der Ressource-Identifier auf, weil Letztere von den Build-Tools durch simple Durchnummerierung der Ressourcen pro Typ erzeugt werden (Siehe 2.3). Die Identifier werden während der Kompilation direkt in den Bytecode aller sie referenzierenden Stellen eingebettet, sodass eine spätere Änderung nicht mehr möglich ist.

Um dieses Problem zu lösen, wird der Build-Prozess der Patch-Packages so modifiziert, dass ihre Ressourcen nicht im Standard-Package mit der ID `0x7f` gespeichert werden. Alle Ressource-Ids des Patch-Paketes können so vergeben werden, ohne einen späteren Konflikt mit der Ziel-App befürchten zu müssen. Während des Patch-Deployments muss der Package-Chunk aus dem Patch-Paket in den Table-Chunk der Ziel-App übertragen werden. Zusätzlich müssen noch die String-Pools abgeglichen, sowie die String-Referenzen im kopierten Package-Chunk adaptiert werden.

1 Dexlib2: https://github.com/JesusFreke/smali/tree/master/dexlib2 Für das Parsen und Schreiben des ARSC-Formats wird unter der freien Apache-2.0-Lizenz Code des Projektes ArscBlamer [2] eingebunden.

5.3. Manifest-Patches

Das Patchen des Android-Manifests beginnt mit der Extraktion des Patch-Manifests aus dem Patch-Package. Im nächsten Schritt wird das proprietäre binäre XML-Format in das breiter unterstützte Standard-Format überführt und schließlich in eine Baum-Datenstruktur zur einfacheren Manipulation geparst. Nachdem auch das Android-Manifest der Ziel-App dieselbe Prozedur durchlaufen hat, werden die im

Patch-Manifest kodierten Änderungen nacheinander abgearbeitet. Als Teil dieses Vorgangs werden auch alle Platzhalter-Ausdrücke aufgelöst. Abschließend wird das resultierende Manifest-XML wieder in das binäre Format übersetzt.

6. Evaluierung

Um die Effektivität und Effizienz des implementierten Patching-Frameworks zu evaluieren, wurde ein Beispiel-Patch auf die 100 populärsten kostenlosen Anwendungen von Google Play angewandt. Alle Messungen wurden auf einem Google Pixel 3 mit Android 11 durchgeführt, um zu demonstrieren, wie performant unsere Lösung trotz der stark eingeschränkten Ressourcen eines Mobilgeräts arbeitet.

6.1. Deployment-Geschwindigkeit

Über unser Test-Set von 100 Applikationen benötigte das Patch-Deployment mittels dynamischem Rewriting im Durchschnitt weniger als 12 Sekunden, während statisches Rewriting durchschnittlich 126 Sekunden benötigte. Für das dynamische Rewriting hat hier die Gesamtgröße des ursprünglichen APK-Files den größten Einfluss, während beim statischen Rewriting die Größe der enthaltenen DEX-Files entscheidender ist.

6.2. Paket-Größe

Die Veränderung der Paketgröße durch das Patchen korreliert direkt mit der Größe des Patch-Paketes, weshalb hier keine konkreten Zahlen genannt werden. Durch den Patch-Vorgang selbst entsteht nur minimaler Overhead. Beim dynamischen Rewriting ist dieser konstant, da eine native Bibliothek in das gepatchte APK injiziert wird, während beim statischen Rewriting ein Overhead nur bei gepatchten Konstruktor-Aufrufen zu bemerken ist (siehe 5.1.1).

6.3. Runtime-Performance

Die Auswirkungen des Patchens auf die Runtime-Performance stehen in direkter Abhängigkeit zur Größe der im Patch spezifizierten Ersatz-Methoden. Enthalten sie weniger Instruktionen als die ursprüngliche Implementierung, so ist besonders für statisches Rewriting sogar eine Verbesserung der Performance möglich. Im Unterschied dazu hat dynamisches Rewriting hier deutlich mehr Overhead, unter anderem, da Patches erst zur Laufzeit angewandt werden. Dennoch entstehen keine bemerkbaren Auswirkungen auf die Nutzererfahrung.

6.4. Kompatibilität

Von den 100 getesteten Applikationen erfüllten 92 % (dynamisches Rewriting) bzw. 93 % (statisches Rewriting) nach dem Patchen noch ihre ursprüngliche Funktion. Obwohl der Patchvorgang immer ein installierbares APK-File erzeugte, stürzten einige gepatchte Anwendungen während der Ausführung ab. Einige Abstürze (4) waren hier auf Spezifika des Beispiel-Patches zurückzuführen, 2 ArscBlamer: https://github.com/google/android-arscblamer nicht auf das Patch-Framework selbst. Zwei Abstürze betrafen Applikationen, die zur Überprüfung der Paket-Integrität Signatur-Checks benutzen.

7. Zusammenfassung

Im Rahmen dieses Projektes wurde ein neues Patching-Framework entwickelt, das gezielt Lösungen für bekannte Limitierungen bestehender Ansätze integriert. Zu diesem Zweck haben wir neue domain-spezifische Sprachen entworfen, die das Formulieren einer Vielzahl von Änderungen verschiedener Bestandteile einer Android-Applikation unterstützen. Außerdem unterstützt das entwickelte Patching-Framework zwei verschiedene Technologien zur Modifikation des Programmflusses. In unserer Evaluierung zeigten wir schließlich, dass die entstandene Lösung als effiziente und effektive Basis für weitere Forschung dienen kann.

8. Bibliographie

[1] C. Tumbleson, „Apktool: A tool for reverse engineering Android apk files," 2021. [Online].

Available: https://ibotpeaches.github.io/Apktool/. [Zugriff am 10 12 2021].

[2] rovo89, „Xposed Installer," 19 06 2014. [Online]. Available:

https://repo.xposed.info/module/de.robv.android.xposed.installer. [Zugriff am 10 12 2021].

[3] Lanchon, „DexPatcher: Modify Android applications at source-level in Android Studio," 09 11 2019. [Online]. Available: https://dexpatcher.github.io/. [Zugriff am 10 12 2021].

[4] android-hacker, „VirtualXposed: Use Xposed without root," 02 06 2021. [Online]. Available:

https://github.com/android-hacker/VirtualXposed. [Zugriff am 10 12 2021].

[5] SaurikIT, LLC, „Cydia Substrate: The powerful code modification platform behind Cydia,"

2014. [Online]. Available: http://www.cydiasubstrate.com. [Zugriff am 10 12 2021].

[6] O. A. Ravnås, „Frida: Dynamic instrumentation toolkit," 03 09 2021. [Online]. Available:

https://frida.re. [Zugriff am 10 12 2021].

[7] V. Costamagna und C. Zheng, „ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime," in *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*, 2016.

[8] T. Ki, A. Simeonov, B. P. Jain, C. M. Park, K. Sharma, K. Dantu, S. Y. Ko und L. Ziarek, „Reptor: Enabling API Virtualization on Android for Platform Openness," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[9] B. Davis und H. Chen, „RetroSkeleton: retrofitting android apps," in *The 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[10] J. Urpalainen, „An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors," 09 2008. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5261. [Zugriff am 10 12 2021].