Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-SIT

Seidlgasse 22 / 9, 1030 Wien
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

Inffeldgasse 16a, 8010 Graz
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

ISOLATION VON UNTERSCHIEDEN IN MOBILANWENDUNGEN

Version 1.0 vom 11.12.2019
Johannes Feichtner – johannes.feichtner@a-sit.at

Anwendungen für Mobilplattformen wie Android und iOS werden häufig aktualisiert und durch Updates über die jeweiligen Distributionskanäle bzw. „App Stores" an Geräte ausgeliefert. Was sich in der jeweiligen Anwendung geändert hat, ist dabei üblicherweise anhand der angeführten Kurzbeschreibung ersichtlich. Bei der Behebung von sicherheitsrelevanten Problemen lässt sich bislang kaum nachvollziehen, an welchen Programmstellen Änderungen vorgenommen wurden.

In diesem Projekt soll eine Lösung erarbeitet werden, um den Programmcode zweier Versionen von Anwendungen miteinander vergleichen zu können, um infolge die Unterschiede bzw. durch ein Update geänderten Programmteile feststellen zu können. Angesichts der fundamentalen Unterschiede im Aufbau und der Funktionsweise von Apps unterschiedlicher Plattformen konzentriert sich diese Studie auf Android. Das Ziel ist es, insbesondere im Hinblick auf sicherheitsrelevante Updates mit vergleichsweise geringem Aufwand festzustellen, ob und wie die Probleme behoben werden.

Inhaltsverzeichnis

1. Einleitung

Android und iOS sind in den letzten Jahren zu den beliebtesten mobilen Betriebssystemen avanciert. Die rasante Dynamik in der Entwicklung stets verbesserter Hardware erstreckt sich auch auf den Leistungsumfang von Android und iOS. Es ist mittlerweile etabliert, dass neue Versionen im Jahresrhythmus erscheinen und neben neuer Funktionalität auch bestehende Kernkomponenten radikal umbauen oder gar entfernen. Diese kontinuierliche Abänderung von Schnittstellen, UI-Elementen und weiterer Funktionalität drängt Hersteller von Mobilanwendungen seit jeher dazu, Apps aktuell zu halten, um eine Kompatibilität mit verschiedenen Versionen von Betriebssystemen und Endgeräten zu gewährleisten. Bugfixes, neue Funktionen und die Behebung von Schwächen, die BenutzerInnen in App-Reviews bemängeln, sorgen insgesamt für eine hohe Update-Frequenz.

Die primären Distributionskanäle für Aktualisierungen von Apps sind Google Play für Android und der Apple App Store für iOS. Neben Screenshots, einer Beschreibung, sowie ggf. weiterer Metainformationen können Hersteller bei Updates auch ein sog. „Changelog“ bereitstellen, das angibt, was sich bei der aktuellen Version im Vergleich mit der früheren geändert hat. In welchem Ausmaß und Detailgrad diese Informationen mitgeteilt werden, obliegt mangels Vorhaben der App Stores alleinig den Herstellern. Die in der Praxis bei Aktualisierungen vorzufindenden Änderungshinweise sind daher auch wesentlich vom Willen der Hersteller zu Transparenz und ihrem Willen zu aussagekräftigen Changelogs geprägt. Dieser Umstand ist insbesondere relevant, wenn Aktualisierungen ausgerollt werden, um sicherheitskritische Defizite zu beheben. Manche App-Hersteller, wie etwa Agilebits[1] im Falle der Passwortmanager-App *1Password*, weisen z.B. vergleichsweise offen darauf hin, wenn Sicherheitsprobleme vorliegen und durch ein Update ausgeräumt werden: „*Prevent revealed passwords from being visible to screen capture tools during drag-and-drop. {591}*“. Viel häufiger finden sich in der Praxis jedoch generische Aussagen, die keine handfesten Rückschlüsse darauf erlauben, was tatsächlich in Apps geändert wurde, etwa wenn es heißt: „*Wir haben diverse Optimierungs- und Leistungsverbesserungen durchgeführt, um ihr Onlinebanking-Erlebnis noch angenehmer und komfortabler zu gestalten.*“.

Nachvollziehen zu können, welche Änderungen in Mobilanwendungen tatsächlich vorgenommen wurden, spielt sowohl aus Sicht von BenutzerInnen, als auch im Kontext einer sicherheitskritischen Analyse von Apps eine wesentliche Rolle. Während sich die Relevanz von Changelogs für wenig sicherheitsaffine AnwenderInnen vor allem auf Änderungen in der Usability konzentriert, ist es bei App-Analysen hingegen essentiell, zu verstehen, worin die Unterschiede zwischen den Apps liegen und ob Sicherheitsprobleme bei Updates (korrekt) behoben wurden. Da Änderungsbeschreibungen potentiell unvollständig sind, möglicherweise nicht der Tatsache entsprechende Angaben machen oder womöglich vom App-Hersteller a priori nicht bereitgestellt werden, ist es umso wichtiger und das **Ziel dieses Projekts**, über eine forschungsmethodische Herangehensweise mehrere Versionen von Apps effizient miteinander vergleichen zu können, um Unterschiede auszumachen. Eine praxistaugliche Möglichkeit, eruieren zu können, welche funktionalen Änderungen es zwischen zwei Versionen einer App gibt, würde wesentlich dabei helfen, Änderungshistorien auf ihre Plausibilität hin zu überprüfen und könnte nicht zuletzt Aufwand und Kosten wiederholter App-Analysen senken.

Aus Forschungssicht ist eine Lösung um zwei Apps miteinander vergleichen zu können eine herausfordernde Angelegenheit. Changelogs umfassen typischerweise nur funktionale Änderungen und klammern Anpassungen ohne Verhaltenseffekt, wie etwa umbenannte Variablen, üblicherweise aus. Die Unterscheidung, ob eine Änderung nur stilistischen Charakter hat oder die Ausführung einer App beeinflusst, ist nicht trivial. Diese Ähnlichkeiten und Unterschiede zwischen Apps *manuell* zu untersuchen, ist angesichts der Komplexität und Größe heutiger Apps mit sehr großem Zeitaufwand verbunden. Die weit verbreitete Verwendung von Techniken zur Codetransformation, insbesondere Maßnahmen zur Code-Obfuscation, Substitution von Bezeichnern durch nichtssagende Platzhalter („identifier renaming“) und vom Compiler vorgenommene Verschiebungen von Code-Elementen („shrinking“, „inlining“) erschweren es zudem signifikant, sinnvolle Rückschlüsse aus der Gegenüberstellung von Apps zu ziehen. *Automatisierte* Ansätze, als alternative Herangehensweise, sind üblicherweise darauf getrimmt, eine Ja-/Nein-Antwort zurückzuliefern, ob eine App eine geklonte [1] oder nur minimal geänderte Version einer anderen App („repackaged app“) ist [2].

[1] https://app-updates.agilebits.com/product_history/OPA4

Bestehende Ansätze zur Erkennung geklonter oder neu gepackter Anwendungen entscheiden typischerweise anhand einer Heuristik, die zwar ein klares Resultat liefert, jedoch nicht in der Lage ist, einzelne Codeteile aufzuzeigen, die zwischen zwei Apps ähnlich oder unterschiedlich sind. Die implementierte Programmlogik zur Bemessung von Ähnlichkeiten basiert dabei in den meisten Fällen auf selbst entwickelten Metriken zur Quantifizierung von „Ähnlichkeit“ aufgrund gewisser, vorselektierter Eigenschaften in Code. Selbst wenn diese Bewertungen im Kontext einer gewissen Anwendung tauglich erscheinen, sind sie oftmals nicht reproduzierbar [3] [4]. Deterministische Herangehensweisen um Ähnlichkeiten in Apps zu analysieren, etwa Androguard[2] oder FSquadra [5], lassen hingegen eine Erklärung vermissen, wie und welcher Code in Relation zueinander steht.

1.1. Lösungskonzept

Um der eingangs skizzierten Zielsetzung dieses Projekts Rechnung zu tragen, wird nachfolgend ein innovativer Ansatz zur vergleichenden Analyse von Mobilanwendungen erarbeitet und im weiteren Verlauf dieses Dokuments detaillierter ausgeführt. Um den Einschränkungen bestehender Ansätze zu begegnen, soll eine Methodik entwickelt werden, die durch einen *qualitativen* Vergleich von Mobilanwendungen Ähnlichkeiten und Unterschiede explizit auf Ebene einzelner Codefragmente aufzeigen kann. Anstatt eines auf Heuristik basierenden Vergleichs oder der Herleitung einer Vergleichsmetrik wird eine direkte Gegenüberstellung zweier Versionen von Apps angestrebt.

Durch die fundamentalen Unterschiede in der Struktur von Anwendungen für Android und iOS erscheint eine generalisierende Lösung nicht zweckmäßig. Angesichts der vergleichsweise einfacheren Rückführbarkeit von Android-Anwendungen hin zu einer Code-Repräsentation und dem damit verbundenen Zeitgewinn bei der Entwicklung eines Prototyps wird im Rahmen dieses Projekts der Fokus ausschließlich auf Android-Anwendungen gelegt.

Um Ähnlichkeiten und Unterschiede in zwei gegebenen Apps aufzuzeigen, wird ein mehrstufiger Analyseansatz vorgeschlagen, der folgenden Anforderungen gerecht werden soll:

- Android-Apps können in einer dekompilierten Fassung mitunter tausende von Codeklassen, verteilt in eine weitläufige Dateihierarchie, umfassen. Ein Vergleich jeder Methode einer Klasse mit jeder Methode in allen anderen Klassen ist möglich, jedoch ineffizient. Um Unterschiede schneller ausfiltern zu können, soll die in Apps vorhandene Hierarchie aus Klassen und Ressourcen effizient abgebildet werden, sodass sich Unterschiede in Apps mit möglichst wenig I/O-Operationen auf das Dateisystem, sowie einer minimierten Anzahl an Vergleichen feststellen lassen.

- Gängige Ansätze zur Verschleierung sowie Transformation von Code sollen umgegangen werden, indem sich der Vergleich auf Code-Eigenschaften stützt, die von Compilern und Systemen zur „Obfuscation“ nicht angetastet werden.

- Vergleichsergebnisse sollen in einer Ansicht visualisiert werden, die einer Repräsentation ähnelt, die von Systemen zur Versionierung von Code (Git, Subversion) bekannt ist.

In den weiteren Abschnitten dieses Dokuments wird detaillierter erläutert, wie sich ein effizienter Vergleich von Code und Ressourcen zweier Android-Apps realisieren lässt.

2. Reverse Engineering von Android-Apps

Um Unterschiede zweier Anwendungen möglichst unzweifelhaft feststellen zu können, ist es essentiell, im ersten Schritt eine Repräsentation von Code zu erhalten, die sich für einen Vergleich eignet. Um zu vermitteln, worin hier die Herausforderungen in der Praxis bestehen, wird nachfolgend in kurzen Auszügen auf das Dekompilieren von Android-Anwendungen, die Repräsentation als Smali-Code, sowie den Umgang mit obfuszierten Codefragmenten eingegangen.

[2] https://github.com/androguard/androguard

2.1. Dalvik-Bytecode in Smali-Repräsentation

Anwendungen für Android werden überwiegend in Java oder Kotlin entwickelt. Während des Kompilierens wird stack-basierter JVM-Bytecode übersetzt zu register-basiertem Dalvik-Bytecode, der bei der Ausführung einer App am Gerät von der „Dalvik Virtual Machine interpretiert“ wird. Indem wiederkehrende Zeichenfolge, Methodensignaturen, Codeblöcke und weitere statische Werte durch Platzhalter ersetzt und bei der Ausführung referenziert werden, kann der Dalvik-Compiler Apps in ihrer Größe komprimieren und alle Teile gesammelt in einer Datei namens *classes.dex* speichern.

Das Dekompilieren von Android-Apps funktioniert in der Regel über Tools wie *dex2jar*[3] oder *enjarify*[4], die jeweils Regeln und Heuristiken zur Auflösung von Abhängigkeiten implementieren und damit aus register-basiertem Dalvik-Bytecode, stack-basierten JVM-Code generieren. Da dieser Prozess nicht deterministisch ist, resultiert dies in der Praxis oft in ungültigen oder falschen Codeübersetzungen.

Eine deterministische Alternative ist die Repräsentation von Dalvik-Bytecode als Smali-Code. Obwohl Syntax und Semantik von Smali sehr nahe an Dalvik-Bytecode gehalten sind, ist diese Repräsentation ähnlich wie Java-Code für Menschen lesbar und deterministisch erzeugbar. Insbesondere dieser Umstand ist für die Zielsetzungen dieses Projekts von zentraler Bedeutung, da eine Isolation von Unterschieden in Mobilanwendungen nicht zielführend wäre ohne Vorliegen von deterministisch generierbaren Vergleichsobjekten.

Das Tool *baksmali*[5] disassembliert eine gegebene *classes.dex* zu Smali-Code und erstellt pro Programmklasse einzelne Dateien, die jener Dateihierarchie entsprechen, die auch im Original als Java-Code vorlagen. Wie in den weiteren Abschnitten dieses Dokuments beschrieben, eignet sich diese Ausgabeform auf Dateiebene für effiziente Vergleiche über Hashbäume bzw. „Merkle Trees“.

2.2. Code-Obfuscation

Im Idealfall wäre es möglich, Codefragmente und verwendete Bibliotheken mit hinreichend guter Genauigkeit nur anhand der Namen von Packages, Klassen und Methoden zu unterscheiden. In der Praxis wird Code jedoch oftmals [6] obfusziert, Namen werden durch Platzhalter ersetzt und verunmöglichen so einen trivialen Vergleich. Um dennoch eine schlüssige und genaue Vergleichsmöglichkeit zu schaffen, fokussieren wir uns auf Eigenschaften in Code die (1) deskriptiv sind, möglich selten auftreten, verändert bleiben für semantisch ähnliche Codeblöcke und (2) von Obfuscation oder gängigen Methoden für Code-Transformation nicht erfasst werden können.

Das bei Android beliebteste Werkzeug zur „Obfuscation“ namens ProGuard stellt App-Herstellern 29 Techniken für Code-Optimierungen zur Auswahl. Das in diesem Projekt vorgeschlagene Lösungskonzept schafft es, die meisten davon zu berücksichtigen. Manche der Techniken, die in der Praxis glücklicherweise nur in Ausnahmen angewandt werden, ändern jedoch den Kontroll- und Datenfluss von Code. Eine Isolation von Unterschieden in Code, die auf Ebene von Methoden und Codeblöcken („Basic Blocks“) operiert, ist folglich nicht in der Lage, Übereinstimmungen von Codeblöcken zuverlässig festzustellen. Dies betrifft insbesondere folgende Transformationen:

- **Inlining:** Hierbei löscht ProGuard Methodenaufrufe und ersetzt den Code durch den Code der aufgerufenen Methoden selbst. Typischerweise kommt diese Technik nur bei sehr kleinen oder einmal verwendeten Methoden zur Anwendung, z.B. sog. „Getter“-Methoden.

- **Zusammenfügen von Code („Merging“):** Mit dieser Transformation reduziert ProGuard replizierte Codefragmente auf nur ein Vorkommen und ändert in Methoden Verweise darauf. Dies wiederum beeinflusst den Datenfluss und „verwirrt“ die Feststellung von Unterschieden.

- **Entfernen von Methodenparametern:** Mit dieser Einstellung identifiziert ProGuard nicht verwendete Parameter und entfernt sie aus Methodensignaturen. Unsere Lösung erkennt in derartigen Methoden keine Übereinstimmung, sondern sieht sie stets als „neu hinzugefügt“.

[3] https://github.com/pxb1988/dex2jar
[4] https://github.com/Storyyeller/enjarify
[5] https://github.com/JesusFreke/smali

3. Analyseablauf

Im Zuge dieses Projekts wird ein Framework entworfen, um Ähnlichkeiten und Unterschiede zwischen zwei gegebenen Android-Anwendungen festzustellen. Der grundsätzliche Analyseablauf lässt sich dabei in folgende zwei Schritte untergliedern:

1. In der ersten Phase findet ein **Vergleich der Ressourcen** statt, die in Apps enthalten sind. Hierfür werden zunächst alle statischen Inhalte, die aus dem Programmcode referenziert werden, wie etwa Bitmaps, Layout-Definitionen und GUI-Strings aus den Apps extrahiert, während die Verzeichnisstruktur in der die Ressourcen vorkommen, unverändert beibehalten wird. Alle Dateien werden daraufhin in die Struktur eines „Merkle Tree“ [6] übertragen und Hashsummen gebildet. Nach Berechnung der Hashwerte von Ordnern und Klassen lassen sich die Baumstrukturen zweier gegebener Apps einander gegenüberstellen und jene Äste entfernen, die in beiden Apps gleich sind. Durch einen Vergleich der Baumstrukturen zweier Apps ist es folglich möglich, effizient herauszufinden, welche Dateien und Ordner mit Ressourcen sich geändert haben, hinzugefügt oder gelöscht wurden.

   Da Ressourcen von „Obfuscation“ oder verwandten Transformationstechniken nie betroffen sind, kann die Feststellung von Unterschieden bereits nach diesem Schritt enden. Zwei Anwendungen mit identischen Ressourcen können darauf hinweisen, dass es sich um geklonte oder „neu gepackte“ Versionen handelt [6] [7].

2. Im zweiten Schritt erfolgt ein **Vergleich des Programmcodes**. Dazu wird der Dalvik Bytecode beider Android-Anwendungen zunächst zu Smali-Code übertragen. Ähnlich wie im ersten Schritt ist der Ausgangspunkt für die weitere Analyse eine Hierarchie von Dateien und Ordnern, die als „Merkle Tree“ organisiert werden. Angesichts der Vielzahl an Programmklassen in heutigen Anwendungen, nutzen wir die Charakteristika dieser Datenstruktur, um in beiden Apps vorkommende Klassen und Packages mit einer minimalen Zahl an Vergleichsoperationen zu eruieren.

   Für die in den Merkle Trees als unterschiedlich identifizierten Codeklassen ist eine weitere Segregation vonnöten. Dazu identifizieren wir alle Methoden, zusammengehörige Gruppen von Code („Basic Blocks“) in Klassen und indizieren sie. In ansteigendem Detailgrad werden daraufhin die in den Apps vorkommenden Elemente wiederholt miteinander verglichen.

4. Bestimmung der Ähnlichkeit von Code in Android-Apps

Um herauszufinden wieviel und welchen Code zwei gegebene Android-Anwendungen untereinander teilen, vergleichen wir Smali-Code auf der Ebene einzelner Klassen und Methoden. Die Grundidee ist ein sog. Teile-und-herrsche-Ansatz, bei dem Programmteile, die in zwei Apps als unterschiedlich erkannt werden, weiter zerlegt und so lange mit Gegenstücken verglichen werden, bis keine weitere Unterteilung mehr möglich ist. Für die zuletzt als unterschiedlich erkannten Fragmente wird final festgestellt, ob es sich um neu hinzugefügte, geänderte oder gelöschte Codeteile handelt.

Im Zuge eines jeden Zerlegungsschritts reduzieren wir die Anzahl der Bezeichner in Code nach einem vordefinierten Schema und substituieren sie mit Platzhaltern. Diese Herangehensweise ermöglicht einen Analyseansatz, der Codefragmente zunächst immer „exakt“ vergleicht und durch fortlaufende Abstraktion „weniger exakte aber hinreichend genaue“ Vergleiche erlaubt. Diese Strategie alleine reicht jedoch nicht aus, um Code-Modifikationen zu berücksichtigen, die im Zuge von Codetransformationen, etwa durch Anwendung von ProGuard, durchgeführt wurden. Um derartigen Bearbeitungen Rechnung zu tragen, wenden wir vor dem eigentlichen Codevergleich verschiedene Operationen auf den Dalvik-Bytecode beider Anwendungen an. Diese Anpassungen verfolgen das Ziel, die Codestruktur und Bezeichner in beiden Apps so anzuordnen, dass in weiterer Folge nur tatsächliche, vom Hersteller induzierte Änderungen, aufzeigt werden. Semantisch unbedeutende Unterschiede, etwa eine andere Reihenfolge der Methoden in Klassen oder vom Compiler jeweils zufällig neu generierte oder anders angeordnete Labels sollen somit nicht als Unterschiede erkannt werden.

[6] https://de.wikipedia.org/wiki/Hash-Baum

4.1. Vergleich von Klassen

Der Vergleich auf Klassenebene ist geprägt von der Idee, Klassen in mehreren Abstufungen miteinander zu vergleichen. Während die erste Vergleichsrunde noch auf eine exakte Übereinstimmung abzielt, klammert die letzte Runde sämtliche Bezeichner aus und sucht somit nach anderen Klassen, die im Kontroll- und Datenfluss äquivalent sind.

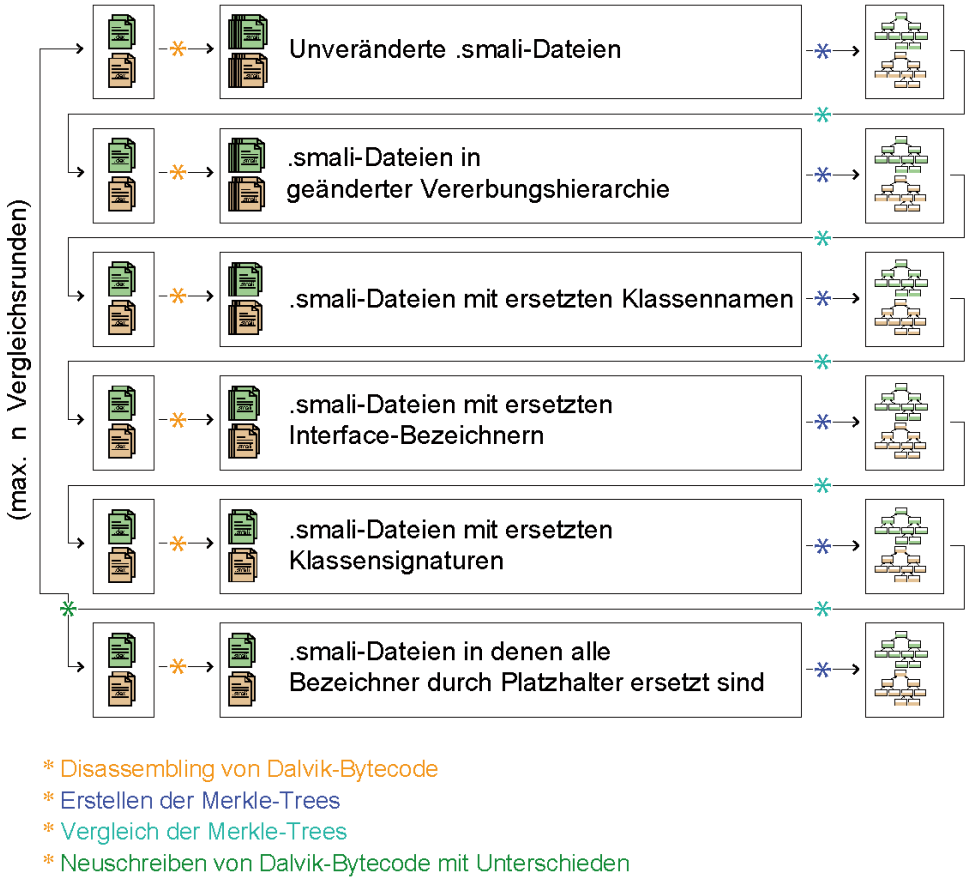

Abbildung 1. Mehrstufiger Vergleich von Klassen.

Der Ablauf dieses aus sechs Iterationen bestehenden Verfahren wird in Abbildung 1 schematisch darstellt. Während der einzelnen Iterationen werden folgende Vergleiche vorgenommen:

1. **Unveränderte Dateien:** Im ersten Schritt werden die Hashwerte der originalen Smali-Dateien verglichen. Unverändert meint in diesem Kontext, dass im Vorfeld keine weiteren Operationen, wie eine Ersetzung durch Platzhalter, darauf angewandt werden. Alle Dateien werden in „Merkle Trees“ angeordnet, die Bäume miteinander verglichen und identische Klassen vor dem nächsten Schritt ausgefiltert.

2. **Dateien in geänderter Vererbungshierarchie:** Im zweiten Schritt werden Klassen gesucht, die vererbt wurden und bei denen sich nur die zugehörige Basisklasse geändert hat. Dazu wird auf Basis des Dalvik-Bytecodes beider Apps ein neues Set von Smali-Code generiert, bei dem Referenzen auf Basisklassen mit Platzhaltern ersetzt werden. Abermals werden Klassen, bei denen eine Übereinstimmung gefunden wurde, nicht mehr weiter betrachtet.

3. **Dateien mit ersetzten Klassennamen**: Nach der Berücksichtigung geänderter Basisklassen im vorigen Schritt, werden nun sämtliche Namen von Klassen durch Platzhalter ersetzt. Dadurch gelingt es, umbenannte Klassen zu finden. Nach diesem Schritt enthält das verbleibende Set von Klassen nur mehr jene Klassen, die entweder ein geändertes bzw. neues Interface implementieren oder eine unterschiedliche Signatur aufweisen.

4. **Dateien mit ersetzten Interface-Bezeichern:** Im vierten Schritt versuchen wir Klassen zu finden, bei denen sich der Namen des zu implementierenden Interfaces geändert hat. Dazu ersetzen wir alle Bezeichner von Interfaces in allen Klassen durch Platzhalter und suchen nach Übereinstimmungen. So wie bei den Schritten wird das Set verbleibender Klassen um jene reduziert, bei denen sich Übereinstimmungen finden.

5. **Dateien mit ersetzten Klassensignaturen:** Für alle Klassen, für die sich bislang keine entsprechende Übereinstimmung finden ließ, wird die Suche nun mit einer Kombination aller vorherigen Bearbeitungen wiederholt. Konkret heißt das, dass in diesem Schritt alle Klassenbezeichner, einschließlich jener von Basisklassen und Interface-Bezeichnern, durch Platzhalter ersetzt werden.

Wie in Abbildung 1 ersichtlich, können die Schritte 1-5 mehrfach wiederholt werden. Zwischen jeder Iteration schreiben wir den Dalvik-Bytecode beider Apps neu und wenden dabei zusätzlich Operationen darauf an, wie z.B. ein Ersetzen der Registernamen durch Platzhalter. Die Schritte können so oft wiederholt werden, solange es in Folgeiterationen zu Übereinstimmungen kommt.

Ein wesentlicher Aspekt, der von den ersten fünf Schritten nicht abgedeckt wird, ist die mögliche Umbenennung von Methoden, infolge derer ein Matching von Klassen verteilt werden würde. Diese kann auch das Resultat einer Bearbeitung durch ProGuard oder ähnlicher Code-Transformationen sein. Um auch diesen Fall abzudecken, ersetzt der letzte Schritt sämtliche Bezeichner durch Platzhalter und klammert so etwaige Umbenennungen aus.

6. **Dateien in denen alle Bezeichner durch Platzhalter ersetzt sind:** Im letzten Schritt entfernen wir sämtliche Bezeichner sowie Namen und substituieren sie durch Platzhalter. Ein Beispiel, wie so eine Ersetzung in der Praxis aussieht, wird in Abbildung 2 veranschaulicht. Bezeichner von Methoden, Feldern und Objekttypen werden durch Platzhalter ersetzt. Spezielle Methodennamen wie `<init>`, gewisse Package-Bezeichner wie z.B. `android/` oder primitive Datentypen werden beibehalten, da sie auch von Obfuscation oder Methoden zur Code-Transformation nicht erfasst bzw. werden können.

```
1  .method public final run()V
2  .registers 3
3
4  iget-object v0, p0, b:Lb/f;
5  iget-object v0, v0, Lb/f;->a:Lb/e;
6  iget-object v1, p0, a:L
       renamed/by/apkcompare/number124;
7
8  invoke-static {v0, v1}, Lb/e;->a
9      (Lb/e;Lrenamed/by/apkcompare/number124;)V
10 return-void
11
12 .end method
```

```
1  .method public final _METHOD_NAME_()V
2  .registers 3
3
4  iget-object v0, p0, _FIELD_NAME_:_TYPE_;
5  iget-object v0, v0, _CLASS_->_FIELD_NAME_:_TYPE_;
6  iget-object v1, p0, _FIELD_NAME_:
       Lrenamed/by/apkcompare/number124;
7
8  invoke-static {v0, v1}, _CLASS_->_METHOD_NAME_
9      (_TYPE_;Lrenamed/by/apkcompare/number124;)V
10 return-void
11
12 .end method
```

Abbildung 2. Vergleich von Smali-Code auf Klassenebene ohne (links) und mit (rechts) ersetzten Bezeichnern.

4.2. Vergleich von Methoden

Der paarweise Vergleich von Klassen ist nicht erfolgreich, wenn Methoden hinzugefügt, geändert oder gelöscht wurden. Für alle Codeklassen, für die im vorigen Schritt keine Übereinstimmung gefunden werden konnte, findet ein detaillierterer Vergleich durch Segregation der Klassen zu einzelnen Methoden und Codeblöcken („Basic Blocks“) statt.

Zunächst werden aus den im vorherigen Vergleich verbliebenen Klassen sämtliche Methoden extrahiert. Für jede von ihnen werden zwei Repräsentationen generiert: eine mit dem unveränderten Code, der sämtliche Bezeichner wie im Original enthält und eine weitere, semantisch äquivalente Fassung, bei der alle Bezeichner durch Platzhalter ersetzt wurden. Naturgemäß enthält die Version, in der originale Bezeichner erhalten wurden, eindeutige Referenzen auf spezifische Klassen und Packages. Die zweite, zur Umgehung von Obfuscation ausgelegte Variante jedoch nicht mehr. Obwohl das Entfernen eindeutiger Referenzen im Lösungskonzept so beabsichtigt ist, um Code-Transformationen zu umgehen, ergeben sich daraus Konsequenzen für die praktische Anwendung.

Wenn die zu vergleichenden Methoden verhältnismäßig klein sind und nur aus wenigen Zeilen Code bestehen, kann ein Methodenvergleich falsche Übereinstimmungen provozieren. Ein Fall, wo dieser Missstand offensichtlich wird, sind sog. „Getter“- und „Setter“-Methoden, die üblicherweise nur aus einer Zeile bestehen. Bei Entfernung der üblicherweise vorkommenden Referenz auf eine zugehörige Klasse, lassen sich diese Methoden, abgesehen von ihrer individuellen Signatur, nicht mehr voneinander unterscheiden. Um dieses Problem in der Praxis zu umgehen, schlagen wir vor, dass eine Methode aus mindestens 5 Zeilen Code bestehen muss, um für Vergleiche herangezogen zu werden. Dieser Wert wurde in Tests empirisch als sinnvolle Untergrenze festgestellt, um mögliche falsche Auffindungen von Übereinstimmungen in den Methoden zweier Apps zu vermeiden. Durch den geringen Informationsgehalt sehr kleiner Methoden sind die Auswirkungen dieses Problems in der Praxis von beschränkter Relevanz.

Für alle Methoden, die die untere Grenze an Codezeilen überschreiten, die für einen sinnvollen Vergleich benötigt werden, berechnen wir einen Hashwert der Methodensignatur, sowie aller Codeblöcke in einer Methode. Dies impliziert, dass die Reihung von Codeblöcken eine Rolle spielt und umgeordnete, verschobene, hinzugefügte oder gelöschte Blöcke das Auffinden übereinstimmender Methoden verunmöglichen. Als Abhilfe für die Fälle, wo sich für Methoden keine gänzliche Übereinstimmung finden lässt, schlagen wir vor, die Suche nach Übereinstimmungen auf die Ebene von Codeblöcken zu erweitern. Der Analyseablauf für Methoden und Codeblöcke lässt sich in vier Vergleichsschritte unterteilen:

1. **Methoden mit Bezeichner:** Nach Extraktion aller Methoden von Klassen, für die bislang keine Übereinstimmung gefunden werden konnte, wird für jede ein Hashwert berechnet und in einer sortierten Liste gespeichert. Methoden bestehend aus weniger als fünf Codezeilen werden ausgefiltert. Für alle verbleibenden Hashwerte werden folglich Übereinstimmungen im Set von Hashwerten der Vergleichs-App gesucht.

   In diesem Schritt gelingt es somit, alle Methoden zu finden, die inhaltlich nicht verändert, jedoch vom Hersteller von einer Klasse in eine andere verschoben wurden.

2. **Methoden ohne Bezeichner:** Für alle Methoden, für die im ersten Schritt keine Übereinstimmung gefunden werden konnte, etwa weil Code-Transformationen angewandt wurden, die Bezeichner nach Zufallsprinzip verändert haben, wird in diesem Schritt eine Repräsentation verwendet, die gegen Obfuscation invariant ist.

   Dieser Schritt findet alle Methoden, die den gleichen Kontroll- und Datenfluss aufweisen, jedoch potentiell unterschiedliche Bezeichner verwenden. In der Praxis ist dies der Fall, wenn etwa eine Version einer Android-App obfusziert ist und das Vergleichsobjekt nicht.

3. **Codeblöcke mit Bezeichner:** Sollte sich der Inhalt von Methoden geändert haben, ist ein Vergleich einzelner Codeblöcke vonnöten. Ähnlich wie im ersten Schritt werden zunächst die Hashwerte aller Codeblöcke berechnet, für die bislang keine Übereinstimmung gefunden wurde. Sie werden danach mit den Hashwerten der Codeblöcke einer gegebenen Vergleichsanwendung gegenübergestellt. Sollten Codeblöcke in mehreren Methoden vorhanden sein, wird eruiert, ob in der zugehörigen Methode auch weitere Codeblöcke übereinstimmen. Diese Entscheidungslogik folgt jenem Schema, das analog auch in Systemen zur Versionierung von Quellcode, wie Git oder Subversion, implementiert ist.

   Im dritten Schritt werden somit alle Codeblöcke gefunden, die von einer Methode in eine andere verschoben wurden, ohne, dass sich Bezeichner geändert haben.

4. **Codeblöcke ohne Bezeichner:** Abschließend werden die Hashwerte der Codeblöcke verglichen, bei denen sämtliche Bezeichner durch Platzhalter ersetzt wurden.

   Der letzte Schritt kommt bei allen gelöschten und neu hinzugefügten Blöcken zur Anwendung. Durch den Vergleich ohne Bezeichner werden außerdem verschobene Codeblöcke gefunden, die nun geänderte Bezeichner, etwa andere Klassen, referenzieren.

Die beschriebene Vergleichsstrategie „wandert“ von der Ebene einzelner Methoden, bis hin zu einem Detailvergleich auf Ebene von Codeblöcken. Angesichts der Tatsache, dass heutige Android-Apps oft tausende von Methoden beinhalten, ist der in diesem Projekt vorgeschlagene Analyseansatz darauf ausgerichtet, die Zahl an Vergleichsobjekten in jeder Etappe schrittweise zu reduzieren. Neu hinzugefügte oder gelöschte Codeblöcke durchlaufen somit immer alle Schritte, da ihre Hashwerte mit jenen aller anderen Blöcke verglichen werden, solange bis schließlich feststeht, dass gesuchte Teile nur in einer Version der gegebenen zwei Anwendungen enthalten sind.

5. Fallstudie

Die nachfolgende Studie zweier Anwendungen veranschaulicht den praktischen Mehrwert der in diesem Projekt geschaffenen Vergleichslösung. Exemplarisch werden durch Updates eingeführte Änderungen in der Kommunikationsanwendung *Skype*[7] sowie dem Passwortmanager *1Password*[8] dem jeweils angegebenen Changelog gegenübergestellt. Es wird somit überprüft, ob das Changelog vollständig ist und inwiefern vom Hersteller gemachte Angaben zu den Änderungen mit den im Code vorgefundenen Anpassungen übereinstimmen. Für beide Anwendungen ist der Quellcode nicht öffentlich, d.h. lediglich die in diesem Projekt geschaffene Lösung und das vom Hersteller gegebene Changelog können für eine Evaluierung herangezogen werden. Für Demonstrationszwecke wurden Updates mit ihren Vorversionen verglichen, die sicherheitskritische Probleme lösen sollen.

5.1. 1Password

Mit über 1 Mio. angegebenen Installationen gehört die App 1Password zu den beliebtesten Android-Anwendungen, um Passwörter auf dem Mobilgerät zu verwalten. Neben kurzen Changelogs, die bei Aktualisierungen via Google Play angeführt werden, listet die Webseite des Herstellers detailliert auf, welche Änderungen zwischen mehreren Versionen vorgenommen wurden[9].

In Version 6.4.1 wurden mehrere sicherheitskritische Probleme behoben, die in früheren Versionen noch präsent waren. Um herauszufinden, welche Änderungen seitens des Herstellers tatsächlich vorgenommen wurden, wenden wir unser Analysekonzept auf Version 6.4 (Build 58) und Version 6.4.1 (Build 59) der 1Password-Anwendung an. Gemäß Herstellerangaben wurden folgende funktionale Änderungen vorgenommen:

- *„Verbesserte Prüfung von URLs“*:
  In früheren Versionen wurden Subdomains bei Eingaben von Benutzer-Innen nicht berücksichtigt, da ein regulärer Ausdruck fehlerhaft war.

  Die mit Version 6.4.1 eingeführten Änderungen ersetzen die Prüfung durch eine neue hinzugefügte Methode namens `getLoginsForUrl`, die unser Vergleichswerkzeug in der Klasse `Utils` vorgefunden hat. Eine Untersuchung des hinzugefügten Codes, der auch in Abbildung 3 dargestellt wird, zeigt, dass die Änderungen das zuvor bestehende Problem tatsächlich beheben.

```
@@ diff: com/agilebits/onepassword/support/Utils.java
     <->
@@       com/agilebits/onepassword/support/Utils.java
   ...

+  public static List<GenericItemBase> getLoginsForUrl(
+    List<GenericItemBase> paramList, String
     paramString)
+  {
+    paramString = PublicSuffix.registrableDomainForUrl
     (paramString);
+    ArrayList localArrayList = new ArrayList();
+    if ((paramList != null) && (!TextUtils.isEmpty(
     paramString)))
+    {
+      paramList = paramList.iterator();
+      while (paramList.hasNext())
+      {
+        GenericItemBase localGenericItemBase = (
     GenericItemBase)paramList.next();
+        if ((!TextUtils.isEmpty(mLocation)) &&
+          (paramString.equals(PublicSuffix.
     registrableDomainForUrl(mLocation)))) {
+          localArrayList.add(localGenericItemBase);
+        }
+      }
+    }
+    return localArrayList;
+  }

   ...

+  public static URI parseURIFromUrl(String paramString
     )
+  {
+    ...
+    return createURIFromUrlStr("https://" +
     paramString);
+  }
```

Abbildung 3. 1Password mit neu hinzugefügter Methode `getLoginsForUrl`.

[7] https://play.google.com/store/apps/details?id=com.skype.raider
[8] https://play.google.com/store/apps/details?id=com.agilebits.onepassword
[9] https://app-updates.agilebits.com/product_history/OPA4

- *„HTTPS statt HTTP“:*
  In älteren Versionen hat der in die App integrierte Browser URLs, bei denen das Protokoll nicht explizit mit HTTPS spezifiziert war, über HTTP aufgerufen. Wie im unteren Teil von Abbildung 3 ersichtlich, wurde Code hinzugefügt, der übergebenen URLs ein „https://“-Präfix voranstellt.

- *„Kein Zugriff auf Nicht-Web-URLs“:*
  In App-Versionen vor dem Update konnten BenutzerInnen auf Daten im Verzeichnis der App zugreifen, indem im internen Browser der Anwendung eine Datei mithilfe des „file://“-Präfix aufgerufen wurde. Wie in Abbildung 4 dargestellt, führt das App-Update eine Beschränkung ein, welche URLs aufgerufen werden dürfen und gibt für alle nicht-konformen URLs eine Fehlermeldung zurück.

- *„Informative Dialoge bei TLS-Fehlern“:*
  Gemäß dem Changelog wurden im Zuge des Updates zu Version 6.4.1 auch die Nachrichten aktualisiert, die BenutzerInnen angezeigt werden, wenn es beim TLS-Verbindungsaufbau zu Fehlern kommt. Eine Analyse der Klasse `CommonWebViewClient` und der String-Konstanten, die den App-Ressourcen (ersichtlich in Abbildung 5) hinzugefügt wurden, zeigt, wie diese Änderung in der Praxis umgesetzt wurde.

```
@@ diff: com/agilebits/onepassword/activity/AutologinActivity.java <->
@@       com/agilebits/onepassword/activity/AutologinActivity.java
   ...
   public void loadUrl(String paramString)
   {
-    paramString = Utils.uriFromUrl(paramString);
+    paramString = Utils.parseURIFromUrl(paramString);
     if (paramString != null) {
-      mWebView.loadUrl(paramString.toString());
+      mWebView.loadUrl(paramString.toASCIIString());
+      return;
     }
+    ActivityHelper.getAlertDialog(this, 2131231548, 2131231547).show();
   }
   ...
```

Abbildung 4. Geänderte Überprüfung bei URLs.

```
@@ diff: values/strings.xml <->
@@       values/strings.xml
   ...
   <string name="UploadedTotalFilesMsg">Uploaded %1 files</string>
   <string name="UploadingFilesIntoKeychainMsg">Uploading files into vault</string>
-  <string name="UrlHint">http://www.example.com/</string>
+  <string name="UrlHint">https://www.example.com/</string>
   <string name="UseDefaultKeychainMsg">Select</string>
   <string name="UseFingerprint">Fingerprint Unlock</string>
  ...
   <string name="recommend_appUrl">market://details?id=com.agilebits.onepassword</string>
   <string name="security_pref_key">security</string>
+  <string name="ssl_date_invalid">The date of the certificate is invalid.</string>
+  <string name="ssl_error_msg">"%1$s couldn't be loaded because of an SSL error."</string>
+  <string name="ssl_error_title">Unable to load page</string>
+  <string name="ssl_expired">The certificate has expired.</string>
+  <string name="ssl_id_mismatch">The certificate hostname does not match.</string>
+  <string name="ssl_invalid">A generic error occurred.</string>
+  <string name="ssl_not_yet_valid">The certificate is not yet valid.</string>
+  <string name="ssl_untrusted">The certificate authority is not trusted.</string>
   <string name="sync_pref_key">sync</string>
   <string name="teams_pref_key">teams</string>
  ...
```

Abbildung 5. Geänderte App-Ressourcen.

Die Fallstudie der Anwendung 1Password zeigt, dass der gewählte Analyseansatz alle vom Hersteller angegebenen Änderungen verifizieren konnte. Die gefundenen Änderungen verdeutlichen außerdem, dass das Changelog vollständig ist und darüber hinaus keine weiteren, sicherheitskritischen Änderungen erfolgt sind, die womöglich nicht erfasst wurden. Die in den Abbildungen präsentierte Darstellungsform, die hinzugefügte Zeilen grün markiert und geänderte bzw. gelöschte rot, ist angelehnt an die Anzeige von Unterschieden in Werkzeugen zur Verwaltung von Quellcode, wie Git oder Subversion. Für BenutzerInnen dieser Systeme ist somit intuitiv gut erfassbar, wie sich zwei Versionen von Mobilanwendungen voneinander unterscheiden.

5.2. Skype

Im Jahr 2011 wurde für die Skype-App über Google Play ein Update bereitgestellt, das die Versionsnummer von 1.0.0.831 auf 1.0.0.983 angehoben hat. Die Aktualisierung sollte ein Problem bereinigen, über u.a. auch medial[10] [11] berichtet wurde. Obwohl der berichtete Mangel bereits viele Jahre zurückliegt, wurden diese beiden Skype-Versionen für einen Vergleich herangezogen, da sie auch bei ähnlichen Analyseansätzen, etwa von Desnos et al. [9], repräsentative Testobjekte waren.

[10] https://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/
[11] https://www.theregister.co.uk/2011/04/15/skype_for_android_vulnerable/

In der verwundbaren Version von Skype wurden vertrauliche Profildaten, wie etwa der Kontostand des Skype-Kontos, das Geburtsdatum, die E-Mail-Adresse, sowie weitere Informationen mit falschen Zugriffsrechten am Android-Dateisystem abgelegt. Die Konsequenz war, dass beliebige Apps Dritter auf diese Dateien zugreifen und die Daten auslesen konnten.

Durch Anwendung unseres Analysewerkzeugs auf beide Versionen der Skype-App konnte herausgefunden werden, dass das Update sicherheitsrelevante Änderungen in der Klasse `com/skype/ipc/SkypeKitRunner` vorgenommen hatte. Die für die Behebung der Schwachstelle wesentlichsten Code-Zeilen wurden in Abbildung 6 zusammengefasst visualisiert.

```
@@ diff: com/skype/ipc/SkypeKitRunner.smali <-> com/skype/ipc/SkypeKitRunner.smali
  ...

  .end method

+ .method private fixPermissions([Ljava/io/File;)V
+   .registers 7
+
+   array-length v0, p1
+
+   ...
+
+ .end method
+
+ .method private chmod(Ljava/io/File;Ljava/lang/String;)Z
+   .registers 7
    ...
    const-string v6, "csf"
-   const/4 v7, 0x3
+   const/4 v7, 0x0

    invoke-virtual {v4, v6, v7}, L
          android/content/Context;->
     openFileOutput(Ljava/lang/String;I)L
          java/io/FileOutputStream;

    ...

    invoke-direct {v2}, Ljava/lang/StringBuilder;-><
          init>()V

-   const-string v4, "chmod 777 "
+   const-string v4, "chmod 750 "

   invoke-virtual {v2, v4}, Ljava/lang/StringBuilder;->
   append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    ...

    move-result-object v1
+   move-object/from16 v3, p0
+
+   iget-object v3, v3, mContext:L
      android/content/Context;
+   move-object v2, v3
+
+   invoke-virtual {v2}, Landroid/content/Context;->
      getFilesDir()Ljava/io/File;
+   move-result-object v2
+
+   invoke-virtual {v2}, Ljava/io/File;->listFiles()[L
      java/io/File;
+   move-result-object v2
+
+   move-object/from16 v3, p0
+   move-object v18, v2
+
+   invoke-direct {v3, v18}, fixPermissions([L
      java/io/File;)V

    invoke-static {}, Ljava/lang/Runtime;->getRuntime()
          Ljava/lang/Runtime;

    ...
```

Abbildung 6. Update der Skype-App mit sicherheitsrelevanten Änderungen.

Wie im oberen Teil des Quellcode-Listings erkennbar, wurden zwei neue Methoden hinzugefügt. Beide sind dazu gedacht, Zugriffsberechtigungen für Dateien, die mit früheren Versionen von Skype erstellt wurden, richtigzustellen. Eine weitere Änderung betrifft den zweiten Parameter, der beim Aufruf der API-Methode `openFileOutput(String, int)` übergeben wird. Mit Hinblick auf die Dokumentation von Android[12] weist diese Änderung darauf hin, dass der Modus um auf Dateien zuzugreifen, von `MODE_READABLE` (3) auf die sicherere Alternative `MODE_PRIVATE` (0) geändert wurde.

Im zweiten Teil des Code-Listings ist ersichtlich, dass Unix-Dateiberechtigungen von 777 auf 750 umgesetzt und im weiteren Verlauf die neu hinzugefügte Methode `fixPermissions` aufgerufen wird.

Zusammenfassend zeigt die Fallstudie der Skype-App, dass unser Analysewerkzeug in der Lage ist, alle Änderungen bei Code und Ressourcen aufzuzeigen, die im Zuge der Behebung eines sicherheitskritischen Problems durchgeführt wurden. Im Hinblick auf das vom Hersteller bereitgestellte Changelog bedeuten die nunmehr identifizierten Unterschiede zweier Versionen der Skype-App, dass die angegebenen Änderungen den beschriebenen entsprechen. Sie zeigen außerdem, dass das Changelog vollständig ist und beim Code und den Ressourcen keine Änderungen vorgenommen wurden, die über die beschriebenen hinausgehen.

5.3. Zusammenfassung der Ergebnisse

Mit der Motivation, die Behebung sicherheitskritischer Probleme in Android-Apps nachvollziehen zu können, wurden exemplarisch jeweils zwei Versionen der Apps 1Password und Skype verglichen. In den vorliegenden Fällen fanden sich in den festgestellten Änderungen keine Diskrepanzen zu den in den Changelogs beschriebenen. Im Umkehrschluss waren die angegebenen Changelogs vollständig und in ihren Angaben vergleichsweise konkret.

[12] https://developer.android.com/reference/android/content/Context.html#openFileOutput(java.lang.String,int)

6. Fazit

Im Zuge dieses Projekts wurde der Fokus auf die Isolation von Unterschieden in Mobilanwendungen gelegt. Das Ziel bestand darin, ein praxistaugliches Lösungskonzept zu erarbeiten, um effizient feststellen zu können, welche Änderungen hinsichtlich Code und Ressourcen eine Anwendung im Zuge eines Updates erfahren hat. Angesichts der fundamentalen Unterschiede im Aufbau und der Funktionsweise von Apps unterschiedlicher Plattformen konzentrierte sich diese Studie auf Android.

Bestehende Konzepte zum Vergleich von Android-Apps verfolgen primär das Ziel, geklonte oder neu verpackte Anwendungen zu finden. Die jeweils verfügbaren Ansätze erlauben keine Aussage darüber, anhand welcher Teile von Code oder Ressourcen sich Apps voneinander unterscheiden. Da in der Praxis zudem sehr häufig Techniken zur Verschleierung und Transformation von Code eingesetzt werden, muss ein Lösungskonzept diese Umstände geeignet berücksichtigen, um beim Vergleich zweier Anwendungen sinnvolle Ergebnisse zu liefern.

Nach anfänglicher Erörterung der Problemstellung und Zielsetzung wurden die Anforderungen an eine Lösung zum Vergleich von Apps zusammengefasst. Nach einer kurzen Einführung in Smali-Code und Obfuscation-Techniken wurde ein Analyseansatz vorgeschlagen und im weiteren Verlauf detaillierter erklärt, der es ermöglichen soll, trotz Anwendung von Code-Transformationen Unterschiede bei Code und Ressourcen zweier gegebener Android-Anwendungen möglichst unzweifelhaft und effizient aufzufinden.

Das in diesem Projekt entwickelte Konzept kann eingesetzt werden, um Änderungen in Apps nachvollziehen zu können, Changelogs von Herstellern auf Plausibilität zu prüfen und neue, durch Updates verursachte Probleme zu identifizieren. Dies unterstützt die sicherheitsorientierte Analyse von Mobilanwendungen und hilft zu verstehen, inwiefern sich Apps voneinander unterscheiden.

Literaturverzeichnis

[1] H. Wang und Y. Guo, „WuKong: a scalable and accurate two-phase approach to Android app clone detection," *Proceeding ISSTA 2015 Proceedings of the 2015 International Symposium on Software Testing and Analysis,* 2015.

[2] C. Jian und M. Alafi, „Detecting Android Malware Using Clone Detection," *Journal of Computer Science and Technology,* 2015.

[3] X. Zhan, Z. Tao und T. Yutian, „A Comparative Study of Android Repackaged Apps Detection Techniques," *Conference: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER),* 2019.

[4] G. Quanlong, H. Heqing, L. Weiqi und Z. Sencun, „Semantics-Based Repackaging Detection for Mobile Apps," *International Symposium on Engineering Secure Software and Systems (ESSoS),* 2016.

[5] Z. Yury und G. Olga, „FSquaDRA: Fast Detection of Repackaged Applications," *28th IFIP Annual Conference on Data and. Applications Security and Privacy (DBSec),* 2014.

[6] D. Wermke, N. Huaman und Y. Acar, „A Large Scale Investigation of Obfuscation Use in Google Play," *Proceeding ACSAC '18 Proceedings of the 34th Annual Computer Security Applications Conference ,* 2018.

[7] S. Mingshen, M. Li und J. C. Lui, „DroidEagle: Seamless Detection of Visually Similar Android Apps," *8th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '15.,* 2015.

[8] Y. Shao und X. Luo, „Towards a scalable resource-driven approach for detecting repackaged Android applications," *Proceeding ACSAC '14 Proceedings of the 30th Annual Computer Security Applications Conference,* 2014.

[9] A. Desnos, „Android: Static Analysis Using Similarity Distance," *45th Hawaii International Conference on System Sciences,* 2012.