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

FINGERPRINTING VON CODE MOBILER APPLIKATIONEN

Version 1.1 vom 05.10.2017
Johannes Feichtner – johannes.feichtner@a-sit.at

Bei der Analyse von Applikationen für mobile Plattformen (Android, iOS) hat sich gezeigt, dass sicherheitsrelevante Probleme oft nicht im Applikationscode selbst zu verorten sind, sondern durch Komponenten von Drittherstellern eingebracht werden. Oftmals werden diese problematischen Codeteile frei zur Verfügung gestellt und finden sich somit in vielen Applikationen wieder. Wird der Programmcode vom Hersteller zudem verschleiert („obfuscated"), erschwert dieser Umstand die Auffindung prekärer Codeteile obendrein.

Das Ziel dieses Projekts bestand darin, eine Strategie zu erarbeiten, um bereits bekannte, im Sicherheitskontext als problematisch bzw. potentiell gefährlich erachtete, Codefragmente automatisiert in Applikationen wiederzufinden. Der so entworfene Ansatz kann dabei helfen, den Aufwand für die manuelle Inspektion von Applikationen zu reduzieren und mitunter das schnelle Auffinden von sicherheitsrelevanten Fehlern trotz Code-Verschleierung signifikant zu erleichtern. Bei der Entwicklung des Konzepts wurde auch auf die praktische Umsetzbarkeit Wert gelegt; repräsentative Ergebnisse aus dem produktiven Einsatz stehen jedoch noch aus.

Inhaltsverzeichnis

1. Einleitung

Smartphones, Tablets und „Wearables“ sind im Alltag präsenter denn je und übernehmen auch zunehmend komplexere Aufgaben. Die dahinter stehende Technologie besticht durch eine Vielzahl an Sensoren, umfassender Rechenleistung und Speichervermögen. Die Entwickler von Mobilanwendungen („Apps“) machen sich diese Kapazitäten zunutze und ergänzen ihre Produkte mit vielfältigeren Funktionen. Ein Nebeneffekt dieses Zuwachses an allen Seiten ist die damit einhergehende Komplexitätssteigerung bei der Entwicklung und Wartung von Apps. Programmcode wird immer unübersichtlicher und Abläufe schwerer nachzuvollziehen. Eine Abhilfe stellt für viele Entwickler daher die Verwendung von Programmbibliotheken von Drittherstellern („Third-Party Libraries“) dar. Funktionalität wird sozusagen ausgelagert und von der Hauptanwendung lediglich über vordefinierte Schnittstellen aufgerufen. Entwickler können ihre Anwendungen so z.B. vergleichsweise einfach um Social Media-, Werbung oder Single Sign On-Funktionen ergänzen.

Die Verwendung von Libraries ist jedoch ein zweischneidiges Schwert. Dem einfach geschaffenen Mehrwert in einer Anwendung steht gegenüber, dass Entwickler zumeist nicht wissen (oder auch wissen können), wie entsprechende Funktionalität umgesetzt wurde. Dieser Aspekt ist insbesondere relevant, wenn Libraries die Verarbeitung sensibler Informationen übernehmen oder mit personenbezogenen Daten arbeiten. In diesem Kontext wurde in mehreren Studien [1, 2, 3, 4] demonstriert, dass sicherheitsbedenkliche Praktiken in Libraries überaus prävalent sind. Die meisten gefundenen Fälle beziehen sich dabei auf „Privacy leaks“, also die unberechtigte Weitergabe personenbezogener Daten an Dritte. Typischerweise geschieht dies durch missbräuchliche Verwendung der Berechtigungen, die einer App zum Installationszeitpunkt erteilt wurden. Medial bekannt geworden ist solches Verhalten beispielsweise in den Libraries für Werbung von Taomike und Baidu, in welchen SMS an Dritte gesandt[1] oder Hintertüren auf den Geräten[2] geöffnet wurden. Verbreiteter als offensichtlich vorsätzlich eingeführte Sicherheitsprobleme finden sich jedoch auch Schwachstellen in Libraries, die potentielle Angriffe begünstigen.

Ausgenutzte Schwachstellen in Libraries können mitunter dazu führen, dass unberechtigterweise Daten mitgelesen[3], zusätzlicher Code in Anwendungen injiziert [5] oder Zugangsdaten gestohlen werden[4]. Selbst nach Bekanntwerden und Behebung von Schwachstellen in Libraries perpetuieren sich die Probleme durch Apps, deren Entwickler keine Updates mit gepatchten Libraries bereitstellen. Ein bekanntes Beispiel hierfür ist etwa die Apache Cordova[5] Library, in welcher bis Version 4.0.2 ein Angreifer das Verhalten von Apps manipulieren konnte. Sämtliche diese Library verwendende Applikationen waren implizit angreifbar.

1.1. Problemstellung

Durch die zunehmende Verwendung in Mobilanwendungen und den damit einhergehenden Risiken wurden auch neue Mechanismen entwickelt, um Libraries zu analysieren. Gängige Ansätze tendieren dabei dazu, Libraries in einer abgekapselten Sandbox auszuführen, um dadurch eventuelle „Privacy Leaks“ zu unterbinden [6, 7, 8]. Ungeachtet der Effektivität dieser Lösungen setzen sie voraus, dass eine Library als solche zuvor zweifelsfrei identifiziert wurde. Für diese Identifikation ist „Reverse Code Engineering“, also die Rücktransformation einer Anwendung zu einer Code-Repräsentation unumgänglich. Der hierfür zu investierende Aufwand ist jedoch beträchtlich und bei jeder zu analysierenden App zu wiederholen.

Die eigentliche Crux an der Analyse von Libraries ist die Tatsache, dass sie sich nicht substantiell vom eigentlichen Programmcode unterscheiden. Es liegt also zunächst ein **Identifikationsproblem** vor. Um Libraries zielgerichtet analysieren zu können, bedarf es einer zuverlässigen Möglichkeit, sie vom restlichen Programmcode zu separieren. Diese Absicht wird dadurch erschwert, dass Anwendungen vor der Veröffentlichung verbreitet „Transformationen“ unterzogen werden, um den Aufwand einer Analyse signifikant zu erhöhen. Zu den häufigsten Modifikationen gehört etwa

[1] https://thehackernews.com/2015/10/android-apps-steal-sms.html
[2] https://thehackernews.com/2015/11/android-malware-backdoor.html
[3] http://blog.parse.com/learn/engineering/discovering-a-major-security-hole-in-facebooks-android-sdk/
[4] https://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html
[5] https://cordova.apache.org

„Obufscation“. Dabei werden sämtliche nicht unbedingt notwendigen Debug-Symbole und Identifier aus dem Code entfernt oder umbenannt. Mithilfe von ProGuard[6] etwa werden die Namen von Methoden, Klassen und Packages auf abgekürzte oder nichtssagende Bezeichner geändert. In der Praxis wird so z.B. aus dem Package-Bezeichner „*com.google*“ die Kurzfassung „*c.a*“. Analog dazu werden auch „Shrinking“ oder „Optimizations“ verbreitet eingesetzt. Hierbei werden nicht genutzte Codeteile beim Kompilieren entfernt und Codefragmente auf effiziente Ausführung hin optimiert. Für die Identifikation von Libraries ist der Vorgang hinderlich, da dadurch relevante Indikatoren verloren gehen. Aufgrund dieser Transformationen sind einfache „Pattern Matching“-Verfahren [1, 8] leider ungeeignet, Libraries hinreichend als solche zu identifizieren.

Eine mit dem Identifikationsproblem eng verwandte Herausforderung ist das **gezielte Auffinden bzw. Vergleichen** von Libraries. Sind beispielsweise Schwachstellen nur in gewissen Versionen vorhanden, muss die Identifikation hinreichend granular arbeiten, um die angreifbare Version oder das darin vorkommende Problem möglichst zweifelsfrei zu erkennen. Die dahinterstehende Idee ist beispielsweise, nach gewissen Versionen einer Library in einer Vielzahl von Apps zu suchen und auf diese Weise jene zu finden, die Schwachstellen haben. Obwohl naheliegend, ist es keine technische Notwendigkeit, dass unterschiedliche Versionen einer Library sich nur in Details unterscheiden. Für eine eindeutige Identifikation ist es somit notwendig, sowohl auf einer Makro- als auch auf einer Mikroebene Charakteristika von Code zu inspizieren und sie der konkreten Version einer Library zuzuordnen. Praktisch bedeutet das, dass einzelne Methoden genauso relevant sind, wie ihnen hierarchisch übergeordnete Klassen und Packages.

Zusätzlich zur Erhebung gewisser Merkmale von Code ist die gezielte Suche nach Libraries auch darauf angewiesen, dass es Referenzmerkmale gibt, die eine Library als solche ausweisen. Ein trivialer Ansatz wäre beispielsweise der Aufbau einer Datenbank mit bekannten Libraries. Angesichts der Vielzahl an verfügbaren Libraries wäre sie jedoch leider inhärent unvollständig.

1.2. Zielsetzung

Als Schlussfolgerung aus der zuvor angeführten Problemstellung lässt sich ableiten, dass Libraries nur anhand von Merkmalen identifiziert werden sollen, die invariant gegenüber „Obfuscation“ sind. Praktisch bedeutet das, dass zunächst herauszufinden ist, welche Attribute und Code-Merkmale von gängigen Methoden der „Code-Transformation“ unabhängig sind.

Nach Feststellung von Eigenschaften (im Folgenden als „Features“ bezeichnet) im „Abstract Syntax Tree“, die Code trotz „Obfuscation“ eindeutig identifizieren, stellt sich die Frage, wie sie sich so in Verbindung bringen lassen, dass ein Vergleich möglich wird. Diese Vereinigung an Eigenschaften wird im Weiteren als „**Fingerprint**“ bezeichnet. Da sich naturgemäß etwa Klassen substantiell von Methoden unterschieden, wird es nicht möglich sein, für beide Konstrukte äquivalente Fingerprints zu generieren. Laienhaft ausgedrückt würden dann Äpfel mit Birnen verglichen werden, was wenig aussagekräftig wäre. Als Kompromiss ist es somit naheliegend, Fingerprints von grundlegenden Attributen zu erstellen (etwa invarianten Methodensignaturen) und sie auf höherer Ebene in einer eigenen Fingerprint-Repräsentation zusammenzufassen. Mit der geeigneten Strategie um diese Fingerprints zu vergleichen, könnten auf diese Weise Packages, Klassen und Methoden jeweils getrennt voneinander charakterisiert werden.

Die zentralen wissenschaftlichen Fragen dieses Projekts lassen sich somit wie folgt formulieren:

- Ist es möglich, Libraries von Programmcode zu unterscheiden? Wenn ja, welche Verfahren eignen sich dafür?
- Welchen Einfluss haben Techniken der „Obfuscation“ auf die Identifizierbarkeit von Code?
- Welche Features eignen sich dafür, Codefragmente möglichst exakt zu identifizieren?
- Wie lassen sich die Features zu Fingerprints zusammenfassen, die untereinander vergleichbar sind?

Der verbleibende Teil dieses Dokuments untersucht die angeführten Fragestellungen und verwendet die dabei festgestellten Erkenntnisse zur Entwicklung eines Fingerprinting-Ansatzes.

[6] https://www.guardsquare.com/en/proguard

2. Grundlagen

Im vorherigen Abschnitt wurde festgestellt, dass „Obfuscation“ eine wesentliche Herausforderung bei der Identifikation von Libraries darstellt. Im Folgenden wird der „Build-Prozess“ einer Android-Anwendung vorstellt. Daraus geht hervor, in welcher Relation Libraries zum übrigen Programmcode stehen, sowie an welcher Stelle und in welchem Ausmaß Code-Transformationen in die Morphologie von Anwendungen eingreifen.

2.1. Erstellungsprozess von Android-Anwendungen

Der Code einer Android-App lässt sich unterteilen in Code, der vom Entwickler einer Applikation selbst geschrieben wird und jenem, der von dritter Seite z.B. in Form einer Library beigesteuert wird.

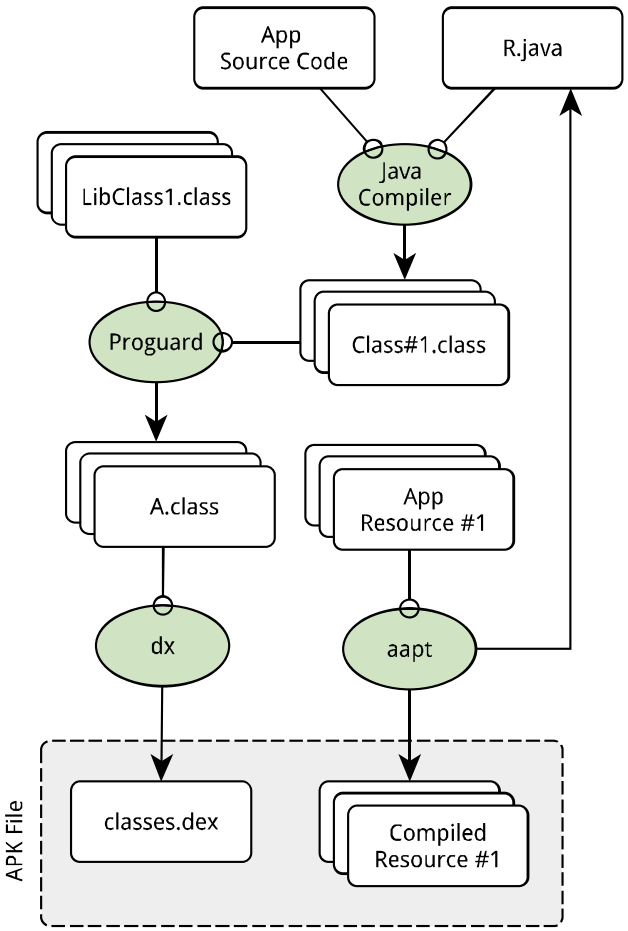

Abbildung 1. Schematischer Ablauf eines Build-Prozesses unter Android[7]

Wie in Abbildung 1 illustriert, sind beim Build-Prozess einer Anwendung mehrere Tools involviert:

1. Der Java-Programmcode einer Anwendung wird zunächst vom Java-Compiler (*javac*) von Quellcode zu stack-basiertem Bytecode umgewandelt.
2. Mithilfe des Tools „Android Asset Packaging Tool“ (*aapt*) wird eine Datei R.java generiert, in welcher sämtliche Ressourcen der Anwendung referenziert werden. Konkret bedeutet das, dass etwa auf beliebige UI-Elemente oder Texte über eine eindeutige Referenz im Code zugegriffen werden kann. Für das in diesem Projekt beabsichtigte Fingerprinting relevant ist die Tatsache, dass diese Ressourcen keiner Transformation untergezogen werden. Ungeachtet etwaiger Code-Transformationen sind also Verweise auf Ressourcen auch invariant gegenüber „Obfuscation“ oder verwandten Transformationsmethoden.

[7] https://developer.android.com/studio/build/index.html#build-process

3. Nach dem Kompiliervorgang von Java-Quellcode zu Bytecode wird das Ergebnis üblicherweise an ProGuard weitergegeben. Unter Verwendung einer vordefinierten Konfiguration wird der Bytecode daraufhin Transformationen untergezogen. Der nachfolgende Abschnitt dieses Dokuments widmet sich den dabei möglichen Schritten.
4. Anschließend wird der Java-Bytecode dem Tool *dx* übergeben, welches die einzelnen Java-Klassen in eine einzelne Datei namens *classes.dex* zusammenfügt und sie dabei zu register-basiertem Dalvik-Bytecode umwandelt.

Im Hinblick auf die Intention dieses Projekts, Attribute aus dem Code von Anwendungen zu extrahieren, um davon Fingerprints zu generieren, ergeben sich mehrere Schlussfolgerungen:

- Ein „Reverse Engineering“ einer Anwendung hin zum originalen Java-Programmcode ist nicht erfolgsversprechend, da durch die Verarbeitung über den Java-Compiler, ProGuard und dx Informationen verloren gehen, die im Endprodukt nicht mehr vorhanden bzw. daraus auch nicht mehr wiederherstellbar sind. Wird der Quellcode anstelle dessen in einer semantisch ähnlichen Approximation rekonstruiert, müsste diese deterministisch sein, um sie für eindeutige Fingerprints verwenden zu können.
- Der Fingerprinting-Prozess an sich könnte entweder auf der Ebene des Dalvik-Bytecodes, des dekompilierten Java-Bytecodes oder einer wiederum dekompilierten Approximation des originalen Quellcodes ansetzen. Ähnlich wie beim Kompiliergang in Vorwärtsrichtung ist aber auch der Dekompiliervorgang nicht frei von Informationsverlust.

Da Dalvik-Bytecode vergleichsweise schwer zu lesen ist, Fingerprints von Java-Bytecode jedoch durch Informationsverlust weniger Aussagekraft haben könnten, stellt die Verwendung einer Zwischenlösung einen sinnvollen Kompromiss dar. Die an Assembly angelehnte Sprache *smali*[8] erfüllt diese Anforderungen und kann Dalvik-Bytecode in einem interpretierbarem Format abbilden, das sich auch als Basis für Fingerprints eignen würde.

2.2. ProGuard

Wenngleich die Weiterverarbeitung von Java-Bytecode über ProGuard keine technische Notwendigkeit darstellt, ist sie ein in der Praxis weit verbreitetes Werkzeug, um Java-Bytecode so zu verarbeiten, dass Syntax und Semantik die Originalfunktionalität nachbilden, nach menschlichem Empfinden aber nicht mehr lesbar sind. Der primäre Einsatzzweck von ProGuard ist also der Schutz geistigen Eigentums, indem Code für menschliche Interpretation unleserlich gemacht wird.

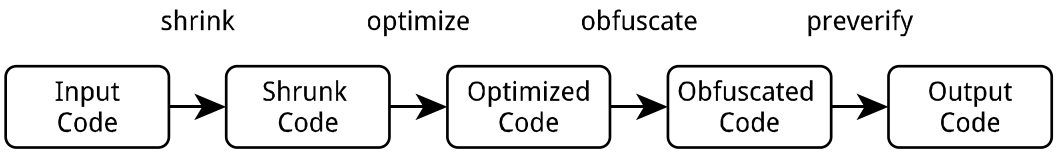

Abbildung 2. Transformationen durch ProGuard

Der prinzipielle Ablauf der Transformationen durch ProGuard ist in Abbildung 2 zusammengefasst. Standardmäßig finden folgende Operationen statt:

1. **Shrinking**
   Im ersten Schritt identifiziert ProGuard unerreichbaren Code mithilfe einer Kontrollflussanalyse und entfernt die entsprechenden Bereiche aus dem Quellcode. Das Ziel ist es, Code auf das zur Ausführung notwendige Minimum zu reduzieren. Dadurch ergeben sich einige Vorteile: Neben einer Reduktion der Anwendungsgröße, benötigt sie auch weniger Speicherplatz am Gerät. Der geringere Umfang führt wiederum zu schnelleren Startzeiten bei der Ausführung[9]. In der Standardeinstellung *„minifyEnabled“* wird Code lediglich um nicht aufgerufene Methoden reduziert. Es darf angenommen werden, dass diese Option dementsprechend oft aktiviert ist. Im Kontext des Fingerprinting bedeutet es, dass Fingerprints auch dann noch aussagekräftig sein müssen, wenn Methoden fehlen sollten.

[8] https://github.com/JesusFreke/smali
[9] https://www.guardsquare.com/en/proguard/manual/examples

2. **Optimization**
   Nach dem Entfernen nicht verwendeter Codefragmente wendet ProGuard Optimierungen an, die das Ziel haben, die Laufzeitperformance zu erhöhen. Zu den eingesetzten Techniken gehören „Inlining“, „Constant Folding“, „Constant Value Propagation“ und sog. „Peephole Optimizations“[10]. In diesem Schritt werden auf feingranularer Ebene (d.h. innerhalb von Methoden) Codefragmente ersetzt, an andere Stellen verschoben, hinzugefügt und entfernt, ohne dabei die Semantik des Codes zu ändern. Dass diese Mechanismen intensiv im Rahmen von Android-Anwendungen angewandt werden, ist prinzipiell nicht anzunehmen. Gleich wie bei Java-Anwendungen werden Optimierungen üblicherweise nicht vom Compiler vorgenommen, sondern finden während der Ausführung durch die Laufzeitumgebung statt. Auch eine Inkompatibilität einiger Optimierungen[11] mit der Android-Laufzeitumgebung lässt davon ausgehen, dass Fingerprints im Normalfall durch Optimierungsschritte nicht wesentlich beeinflusst oder verfälscht werden können.
3. **Obfuscation**
   Unter diesem Terminus versteht man allgemein Mechanismen, die helfen sollen, dass Code möglichst schwer „reverse engineered“ also dekompiliert werden kann. Obwohl es im Allgemein eine Vielzahl unterschiedlicher Ansätze und Techniken gibt, ist die von ProGuard angewandte Technik vergleichsweise naiv. Debugsymbole, wie etwa die Namen von Variablen, Klassen, Methoden und Packages werden entfernt und durch kurze, nichtssagende Zeichen ersetzt. Die Konsequenz daraus sind etwa Package-Namen wie „*a.a.a.a*“ oder „*a.a.a.b“*. Ein praktischer Nebeneffekt des Verschleierns der Semantik für menschliche Interpretation ist die Tatsache, dass sich durch das Abkürzen von Namen auch die Codegröße reduziert. Auch diese Transformation ist standardmäßig aktiv wenn ProGuard mit der Option „*minifyEnabled“* aufgerufen wird. Durch diese Operation wird auch das in Abschnitt 1.1 erwähnte „Identifier Matching“ verunmöglicht. Trivial heißt das, dass eine Google-Library etwa nicht deswegen als solche zu erkennen ist, wenn sie ein Package von „*com.google“* beinhaltet.
4. **Preverification**
   Im letzten Schritt fügt ProGuard Informationen in den Code ein, die sicherstellen sollen, dass der „Class Loader“ bei der Ausführung einer Anwendung keine bösartigen Aktionen vornehmen kann. Da die Android-Laufzeitumgebung diese Angabe aber offenbar nicht nutzt[12], wird diese Funktionalität mangels Relevanz bei Android nicht weiter betrachtet.

Zusammenfassend lässt sich feststellen, dass Fingerprinting also unabhängig davon funktionieren muss, wie einzelne Identifier heißen und wenn Methoden fehlen. Es verbleibt somit nur ein Untersuchungsansatz auf Basis des „Abstract Syntax Tree“ (AST), welcher im Vergleich das höchste Maß an Invarianz gegenüber ProGuard-Transformationen darstellt. Die hierarchische Struktur des AST hat zudem den Vorteil, dass es irrelevant ist, auf welcher Hierarchieebene im Code eine Library vorkommt. Sollte eine Library also als Code eingebettet im Package „*com.lib1“* eingebettet werden, wäre sie gleichermaßen identifizierbar wie unter „*com.pkg.lib1*“. Ein weiteres Alleinstellungsmerkmal des AST ist die Tatsache, dass einzelne Methoden je nach Ausgangspunkt nicht alleine darüber etwas aussagen, ob eine Klasse oder Package zu einer Library gehören. Als Analogie zu verstehen ist beispielsweise, wenn bei einem Baum mehrere Äste (hier: „Shrinking“) weggeschnitten werden würden, wäre u.U. immer noch genügend Material da, um das Wesen des Baums (hier: Library XY) hinreichend zu identifizieren.

2.3. Libraries

Libraries von Drittherstellern werden üblicherweise entweder in den Code einer Applikation eingebettet oder im Rahmen einer *jar*- oder *aar*-Datei gebündelt beigelegt. Die Einbettung in den gewöhnlichen Programmcode erhöht die Komplexität einer Identifikation als solche ungemein, da sich Quellcode von Library-Code unterschieden muss. Wird eine Library hingegen als fertiges Konvolut hinzugefügt, ist sie als solche auch klar abgetrennt und gut erkennbar. Libraries, die im „Java Archive“- oder „Android Archive Library“-Format beigelegt werden, sind nicht von

[10] https://www.guardsquare.com/en/proguard/manual/optimizations
[11] https://stackoverflow.com/questions/35321742/android-proguard-most-aggressive-optimizations
[12] https://www.guardsquare.com/en/proguard/manual/examples#androidactivity

Transformationen durch ProGuard erfasst. Die Identifikation und das Wiederauffinden einer solchen Library ist somit trivial und keine wissenschaftlich relevante Problemstellung. Das Interesse dieses Projekts gilt daher vielmehr jenen Libraries, die gemischt mit gewöhnlichem Quellcode vorkommen.

3. Verwandte Arbeiten

In den letzten Jahren haben sich mehrere wissenschaftliche Arbeiten der Identifikation von Apps gewidmet, die nur mit kleinen Änderungen als neue Apps veröffentlich wurden („repackaged apps“). Die Idee bestand darin, diese Unterschiede möglichst vollständig zu identifizieren und mit einem Pendant zu vergleichen. Dieser Vergleich ähnelt systematisch sehr dem Vergleich von Libraries sowie der Suche von Libraries in Programmcode.

Um „repackaged apps“ zu finden, verwenden entsprechende Arbeiten eigene Fingerprinting-Verfahren, die, ähnlich zu diesem Vorhaben, Apps dekompilieren, Hashing-Verfahren auf Code anwenden und die Hashes z.B. jenen in der Originalversion einer App vergleichen [9, 10, 11]. Alternativ dazu ist ein Vergleich auch über semantische Features möglich, die aus dem „Program Dependency Graph“ extrahiert werden [12, 13].

Schleimer et al. propagieren eine allgemeinere Lösung, die grundsätzlich zur Erkennung von Plagiaten gedacht ist [14]. Der vorgeschlagene Algorithmus unterteilt dabei eine Textdatei in sich überlappende Fragmente, wendet ein Hashing-Verfahren darauf an und vergleicht die einzelne Segmente mit Gegenstücken. Die Lösung zielt rein auf Texte ab, ist agnostisch gegenüber der eingesetzten Sprache und erfordert keine besondere Dokumentstruktur oder Semantik. Die Lösung würde sich also grundsätzlich auch für dekompilierten Quellcode von Android-Anwendungen eignen.

Potharaju et al. haben einen alternativen Ansatz namens „AST Distance“ vorgeschlagen, der sich anstelle des Quellcodes auf den „Abstract Syntax Tree“ fokussiert [15]. Auf Basis extrahierter ASTs können Fingerprints generiert und verglichen werden. Der prinzipielle Ablauf ist dabei wie folgt:

1. Umwandeln des Dalvik-Bytecodes einer Anwendung zu Smali-Code.
2. Code entfernen bis nur mehr relevante Features, wie die nachstehenden, übrigbleiben:
    - Zu jeder Methodensignatur wird die Anzahl der Parameter festgehalten.
    - Jeder Methodenaufruf innerhalb einer Methode über die Instruktionen *invoke-direct* oder *invoke-virtual* verbleiben.
    - Alle Bezeichner von Variablen werden mit Platzhalter ersetzt.
3. Aus dem verbleibenden Konvolut wird ein AST konstruiert.
4. Auf Basis des AST werden Fingerprints erstellt. Dazu wird die Anzahl der verfügbaren Features gezählt und in einen Vektor mit fixer Größe eingetragen. Dieser „Featurevektor“ besteht aus sog. horizontalen und vertikalen Features. Ein horizontales beschreibt das Vorkommnis von Kindknoten mit gleichen Eltern im AST, ein vertikales repräsentiert einen gerichteten Pfad beliebiger Länge im AST.
5. Der Fingerprint einer App wird basierend auf den Fingerprints einzelner Methoden berechnet.

Wie die Berechnung von Fingerprints in der Praxis aussieht, wird in Abbildung 3 veranschaulicht. Um schließlich eine Aussage darüber treffen zu können, ob eine Anwendung der Kopie einer anderen entspricht, verwendet „AST Distance“ die euklidische Distanz zwischen beiden App-Fingerprints und evaluiert, ob die Distanz der beiden einen gewissen Schwellwert überschreitet. Die Basis für diese Erhebung ist die Hypothese, dass zwei Apps ähnlich sind, wenn ihre Fingerprints nahe beisammen liegen.

Im Hinblick auf das Vorhaben dieses Projekts stellt „AST Distance“ einen sehr fortgeschrittenen Ausgangspunkt dar, auf dem aufgebaut werden kann. Der Ansatz weist lt. Autoren nur 0,5% falsche Treffer („false positives“) aus, was wiederum unterstreicht, dass der AST eine konstruktive Ausgangsbasis für Fingerprints ist. Ungeachtet dessen scheint diese Lösung nicht unmittelbar anwendbar zu sein, um Libraries in Android-Anwendungen zu identifizieren und wiederaufzufinden:

- Im zweiten Schritt des Ablaufs (siehe Abbildung 3), reduziert „AST Distance“ den Quellcode auf die elementarsten Instruktionen. Es ist dadurch zwar sichergestellt, dass der Kontrollfluss nach wie vor korrekt abgebildet wird, für eine Identifikation von Libraries ist die gegebene Information aber zu gering.

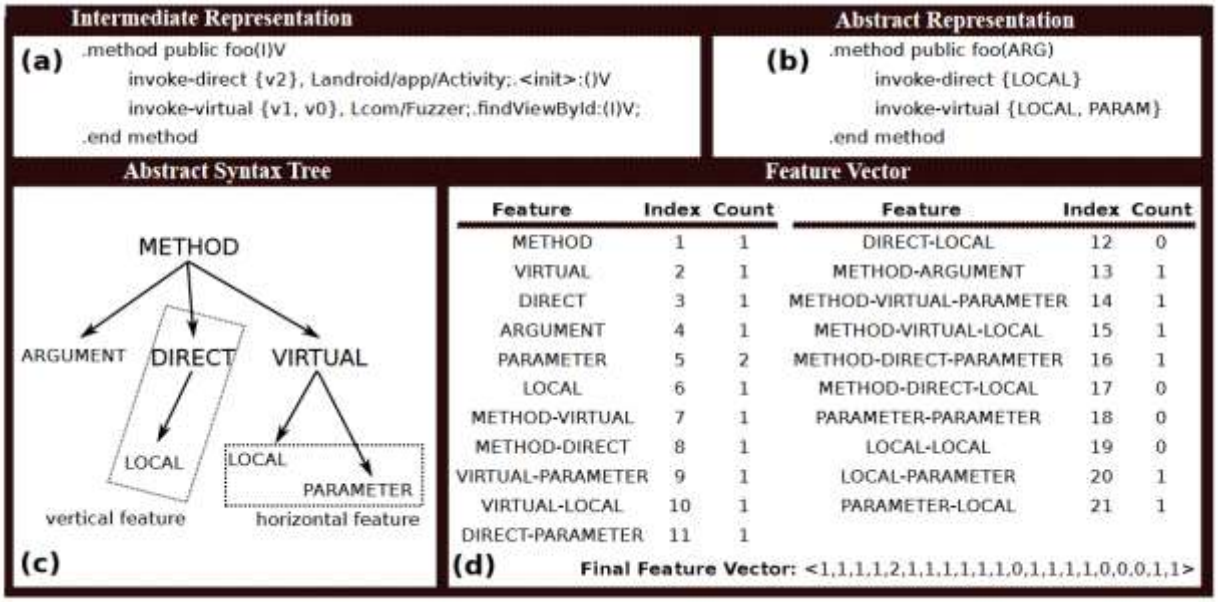

Abbildung 3. Generierung eines Fingerprints auf Basis des AST [15].

Ändert sich z.B. der Kontrollfluss einer Library zwischen einzelnen Version nicht, ist auch eine Unterscheidung in einzelne Versionen einer Library nicht möglich.

- Die Dimensionen des von „AST Distance“ erzeugten Vektors hängen voneinander ab. Beispielsweise sind alle vertikalen Features, die mit *method* beginnen, länger als ein Knoten (z.B. *method-virtual*). Auch die Features *local-param* und *param-local* sind voneinander abhängig, da der Algorithmus außer Acht lässt, in welcher Reihenfolge sie auftreten. Praktisch heißt das, dass insgesamt nur 12 von 21 Dimensionen des Vektors nicht direkt voneinander abhängen – wodurch effektiv 43% eines Vektors ungenutzt bleiben.
- Im letzten Schritt generiert „AST Distance“ einen App-Fingerprint durch Aufsummieren der Fingerprints aller Methoden. Wir können diese Idee für die Identifikation von Libraries übernehmen, indem anstelle der Fingerprints aller Methoden die Fingerprints von Packages und Klassen zusammengefasst werden. Ungeachtet dessen erfordert dieses Vorgehen, dass jede Klasse für die ein Fingerprint berechnet wurde, „vollständig“ ist, d.h. alle Methoden der Klasse präsent sind. In der Praxis ist das unwahrscheinlich, da z.B. die „Shrinking“-Operation von ProGuard ungenutzten Code eliminiert. Alle Methoden zur Berechnung eines Fingerprints der gesamten App bzw. in unserem Fall Library heranzuziehen, würde also aufgrund von „Shrinking“ nicht funktionieren bzw. keine vergleichbaren Fingerprints liefern.

Es lässt sich also feststellen, dass das „AST Distance“-Verfahren von Potharaju et al. einen sehr ähnlichen Zugang verfolgt, jedoch mit dem Ziel, minimal geänderte Kopien von Apps zu finden, und nicht Libraries zu identifizieren. Da ein Mechanismus zu Ableitung und Vergleich von Fingerprints jedoch bereits enthalten ist, stellt der Ansatz eine sehr fortgeschrittene Basis dar.

Ein noch verbleibender Aspekt ist die Frage, wie Libraries als solche konkret erkannt werden können. Li et al. [16] haben in einer ähnlichen Fragestellung eine Liste populärer Libraries zusammengestellt. Um herauszufinden, welche das sind, wurde der Name von vorkommenden Packages in Android-Anwendungen ausgewertet. Naturgemäß funktioniert das nur, wenn keine „Obfuscation“ zum Einsatz kommt und die Package-Bezeichner einem eindeutigen Schema folgen. Eine vordefinierte Liste von Libraries wäre zudem inhärent unvollständig. In der Praxis bedeutet das, dass nur jene Libraries gefunden werden könnten, die anhand ihres Bezeichners vorher klar erkannt wurden. Um auch mit „Obfuscation“ umgehen zu können, haben Backes et al. in ihrer Lösung namens „LibScout“ eine Heuristik umgesetzt, die neben den Bezeichnern auch die Struktur der Pakethierarchie und Methodensignaturen einbezieht. Kommt jedoch „Shrinking“ zum Einsatz, fallen diese Eigenschaften mitunter weg, worin auch die Schwachstelle dieser Lösung liegt. Durch eine Fokussierung auf den AST wäre ein alternativer Ansatzpunkt geschaffen.

4. Fingerprinting über den „Abstract Syntax Tree“

In diesem Abschnitt wird eine Weiterentwicklung des Fingerprinting-Ansatzes von Potharaju et al. vorgestellt (siehe Abschnitt 3). Auf Basis der bestehenden Arbeit wurde dabei der Fokus auf die Identifikation Libraries gelegt und ein Konzept entwickelt, um Libraries zu lernen und eine Übereinstimmung mit beliebigem Quellcode zu prüfen.

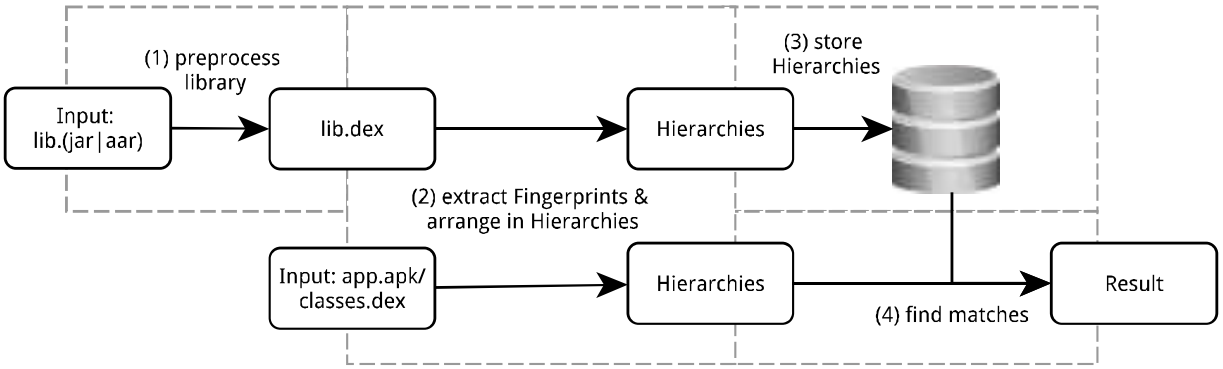

Abbildung 4. Ablaufbeschreibung des Fingerprinting bei Libraries.

Abbildung 4 demonstriert die Ableitung von Fingerprints auf Basis des „Abstract Syntax Tree“ einer gegebenen Library. Bevor eine Library als solche in einer Mobilanwendung erkannt werden kann, steht im ersten Schritt ein Lernprozess an. Dieser bedingt, dass Features einer Library separat von tertiärem Quelltext extrahiert werden. Wie auch von Li et al. [16] festgestellt, ist eine heuristische Erkennung von Libraries ohne vorherige „ground truth“ de facto nicht möglich, da in Apps eingebettete Libraries nicht prinzipiell als solche erkennbar sind. Demzufolge illustriert der obere Teil der Grafik den **Identifikationsprozess**, im Zuge dessen Features extrahiert, zu Fingerprints verarbeitet und jene mit Verweis auf die Library gespeichert werden. Der untere Teil veranschaulicht das **Wiederauffinden** von zuvor identifizierten Libraries in Android-Anwendungen.

Die einzelnen Schritte des Lernens und Wiederauffindens finden in folgendem Ablaufschema statt:

1. Bei der anfänglichen Lernphase wird eine Library im „Java Archive“- oder „Android Archive“-Format (siehe Abschnitt 2.3) zu Dalvik-Bytecode umgewandelt. Wie in Abschnitt 2.1 dargestellt, findet ein analoger Vorgang auch statt, wenn Android-Anwendungen kompiliert werden. Die Transformation von stack-basiertem Java-Bytecode zu Dalvik-Bytecode ist in der Praxis etwa mit dem Tool *dx* möglich, welches im Android SDK enthalten ist und auch bei gewöhnlichen Kompiliervorgängen von Anwendungen eingesetzt wird.

2. Der zweite Schritt betrifft Lernen und Wiederauffinden. Dalvik-Bytecode muss hierfür zunächst in eine Darstellung als Smali-Code gebracht werden. Basierend auf Smali-Code können Features extrahiert und davon Fingerprints generiert werden.

   Basierend auf Smali-Code wird der AST abgebildet und daraus einzelne, für den Fingerprint relevante Features extrahiert. Diese werden daraufhin in einer Vektorrepräsentation angeordnet, um eine Vergleichbarkeit untereinander zu gewährleisten. In Analogie zu Potharaju et al. wird der AST hierfür in einer Breitensuche durchschritten und es werden Instruktionen des Typs *invoke-direct* und *invoke-virtual* isoliert. Da dieser Instruktionstyp in der Regel für Aufrufe von Methoden verwendet wird, gibt es zusätzlich auch eine beliebige Anzahl an Aufrufparametern zu berücksichtigen. Zusammenfassend ergibt sich eine minimalere Form des AST, aus welchem hervorgeht, welcher Methoden von anderen mit welchen Parametertypen aufgerufen werden.

   Die Fokussierung auf die jeweiligen Typen von Methode und Parametern stellt sicher, dass die Erkennung auch unabhängig von „Obfuscation“ funktioniert. Dies ist sichergestellt, da wie in Abschnitt 2.2 erhoben, etwaige Transformationen (z.B. „Identifier Renaming“) sich nicht auf Dateitypen erstrecken. Salopp gesagt heißt das, dass eine Transformation etwa den Typ eines Rückgabewerts oder Parameters nicht ändern kann (und darf). Darauf aufbauend

können die extrahierten Features in einer Vektorrepräsentation angeordnet werden. Für die darauffolgende Generierung von Fingerprints verwenden wir den Ansatz von Potharaju et al.

3. Nach Generieren von Fingerprints für einzelne Methoden und Packages sollen die Ergebnisse in einer Datenbank mit Verweis auf die jeweilige Library abgelegt werden.

   Die auf Basis der Vektorrepräsentation generierten Fingerprints eignen sich in diesem Zustand bereits für Vergleiche einzelner Methoden mittels Berechnung der euklidischen Distanz. Für den Verwendungszweck in verwandten Arbeiten ist dieser granulare Vergleich auf Methodenebene ausreichend. Sollen Libraries jedoch als Konvolut identifiziert werden, wird eine abstraktere Darstellung benötigt. Beispielsweise sind sog. *Getter-* und *Setter-Methoden* zumeist recht ähnlich, was sich auch in ihren Fingerprints widerspiegeln würde. Nichtsdestotrotz müssten sie deshalb nicht zwangsläufig zur gleichen Klasse gehören. Durch eine Zusammenfassung der Fingerprints von Methoden, soll auch ein Vergleich auf Ebene einer Klasse oder eines Packages möglich sein. Zur Abbildung dieser Relation wird eine Datenstruktur, die Relationen in Abhängigkeiten abbilden kann. Eine Möglichkeit hierfür wäre z.B. die Verwendung eines „Merkle Trees“ bzw. Hash-Baum.

4. Wird nach Libraries in Android-Anwendungen gesucht, verfolgt die Suche nach Übereinstimmungen den gleichen Ablauf wie beim Lernen, vergleicht jedoch die Fingerprints, der in der App vorhandenen Packages mit den zuvor gelernten und in der Datenbank vermerkten. Potharaju et al. haben mehrere Methoden aufgezeigt, wie Fingerprints verglichen werden können. Unter den erprobten hat sich herausgestellt, dass die sog. „Manhattan-Distance“ [17] die Unterschiede einzelner Fingerprints am deutlichsten dargestellt hat. Es erscheint daher naheliegend, diesen Ansatz zu übernehmen.

   Da die Datenbank mit Fingerprints sowohl Einträge von Methoden, Klassen und Packages enthält, muss bei der Suche nach bereits bekannten Einträgen ein strukturierter Ansatz gewählt werden. Um jeweils eine Library in ihrem möglichst vollständigen Umfang zu erfassen, wird eine Suche von „außen nach innen“ vorgeschlagen. Bei der Suche nach übereinstimmenden Fingerprints soll in der Reihenfolge: Packages, Klassen, Methoden gesucht werden. Diese hierarchische Suche gestaltet sich im Gesamten also wie folgt:
   a. Sortiere alle Fingerprints in der Datenbank nach ihrer hierarchischen Position.
   b. Suche nach Fingerprints, die den gleichen AST-Vektor abbilden. Entspricht der Fingerprint eines zuvor gelernten Packages jenem in einer App, ist ein schneller Treffer möglich, ohne die Hierarchie bis auf Methodenebene hinunter zu schreiten.
   c. Sollte ein Treffer nicht unmittelbar möglich sein, wird mittels der „Manhattan Distance“ geprüft, welche Packages die ähnlichsten Fingerprints haben. In absteigender Reihenfolge des Ausmaßes der Übereinstimmung sollen daraufhin die Klassen des jeweiligen Packages mit denen in der App verglichen werden. Bei dieser Prüfung ist sowohl möglich, dass in der App zusätzliche Klassen vorkommen, die in der gelernten Library nicht vorhanden sind (z.B. bei einer Library in neuerer Version) bzw. umgekehrt (und häufiger) kann es sein, dass durch „Shrinking“ in der originalen Library mehr Klassen vorhanden sind als in der App. Sollte also eine Übereinstimmung in allen Klassen vorliegen, im hierarchisch übergeordneten Package aber nicht, ist davon auszugehen, dass Klassen durch Code-Transformationen abhandengekommen sind.
   d. Die im Punkt c. beschriebene Suche ist auf jeder Hierarchieebene zu wiederholen. Naturgemäß sinkt die Wahrscheinlichkeit, eine Übereinstimmung gefunden zu haben, auf jeder Ebene. Das heißt, wenn es Übereinstimmungen bei den Fingerprints der Package-Hierarchie gibt, ist es eher plausibel, dass eine Library gefunden wurde, als auf Methodenebene. Wurde eine Library nicht gelernt bzw. sind ihre Fingerprints in der Datenbank nicht bekannt, kann die hierarchische Suche zumindest Methoden, Klassen und Packages mit ähnlichen Fingerprints vorschlagen. Ein ähnliches Szenario trifft zu, wenn Apps z.B. andere Versionen von Libraries enthalten, als die die gelernt wurden. Umgekehrt heißt das, dass nicht notwendigerweise die Fingerprints einer jeden Version einer Library im Vorfeld gelernt werden müssen.

5. Fazit

Im Zuge dieses Projekts wurde ein Konzept erarbeitet, wie Code in Mobilanwendungen identifiziert und wiederaufgefunden werden kann. Da Libraries von Drittherstellern aus Sicherheitsperspektive eine überaus relevante Komponente in modernen Anwendungen sind, wurde ein Fokus auf die Erkennung von Code in Libraries gelegt.

Einen wesentlichen Beitrag dazu, dass dieses Vorhaben gelingen kann, ist die Wahl des richtigen Ansatzes für die Generierung von Fingerprints. In der Problemstellung wurde dargelegt, dass „Obfuscation“ und andere invasive Code-Transformationen verbreitet eingesetzt werden und dementsprechend eine Basis zu finden ist, die hiergegen invariant ist. Die Studie verwandter Arbeiten hat aufgezeigt, dass Fingerprinting auf Basis des „Abstract Syntax Tree“ einer Anwendung die gewünschte Stabilität bietet.

In diesem Projekt wurde der Ansatz von Potharaju et al. so erweitert, dass Fingerprints in einer Hierarchie abgebildet und in Abhängigkeit zueinander gesetzt werden können. Dieses Konzept bietet das Potential, z.B. zielgerichtet verwundbare Libraries in Anwendungen aufzufinden.

Referenzen

[1] M. Grace, W. Zhou, X. Jiang und A.-R. Sadeghi, „Unsafe Exposure Analysis of Mobile In-App Advertisements,“ *WISEC,* 2012.

[2] T. Book, A. Pridgen und D. Wallach, „Longitudinal Analysis of Android Ad Library Permissions,“ *MoST,* 2013.

[3] J. Seo, D. Kim, D. Cho, T. Kim und I. Shin, „FlexDroid: Enforcing In-App Privilege Separation in Android,“ *NDSS,* 2016.

[4] R. Stevens, J. Crussell, C. Gibler und H. Chen, „Investigating User Privacy in Android Ad Lbraries,“ *MoST,* 2012.

[5] S. Poeplau, Y. Fratantonio, A. Bianchi und G. Vigna, „Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications,“ *NDSS,* 2014.

[6] P. Pearce, A. Porter Felt, G. Nunez und D. Wagner, „AdDroid: Privilege Separation for Applications and Advertisers in Android,“ *ASIACCS,* 2012.

[7] S. Shekhar, M. Dietz und D. Wallach, „AdSplit: Separating Smartphone Advertising from Applications,“ *USENIX,* 2012.

[8] W. Yang, J. Li und Y. Zhang, „APKLancet: Tumor Payload Diagnosis and Purification for Android Applications,“ *ASIACCS,* 2014.

[9] K. Chen, P. Liu und Y. Zhang, „Achieving Accuracy and Scalability simultaneously in detecting Application Clones on Android Markets,“ *ICSE,* 2014.

[10] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen und D. Song, „Juxtapp: A scalable System for Detecting Code Reuse among Android Applications,“ *DIMVA,* 2012.

[11] W. Zhou, Y. Zhou, X. Jiang und P. Ning, „Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces,“ *CODASPY,* 2012.

[12] J. Crussell, C. Gibler und H. Chen, „Attack of the Clones: Detecting Cloned Applications on Android Markets,“ *ESORICS,* 2012.

[13] J. Crussell, C. Gibler und H. Chen, „Andarwin: Scalable detection of semanitcally similar Android Applications,“ *ESORICS,* 2013.

[14] S. Schleimer, D. S. Wilkerson und A. Aiken, „Winnowing: Local Algorithms for document fingerprinting,“ *SIGMOID,* 2003.

[15] R. Potharaju, A. Newell, C. Nita-Rotaru und X. Zhang, „Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques,“ *Engineering Secure Software and Systems,* 2012.

[16] L. Li, T. F. Bissyandé, J. Klein und Y. L. Traon, „An investigation into the use of Common Libraries in Android Apps,“ *arXiv preprint 1511.06554,* 2015.