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

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

A-8010 Graz, Inffeldgasse 16a
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

STATISCHE ANALYSE VON IOS-APPLIKATIONEN

Version 1.0, 14. Dezember 2016
Johannes Feichtner – johannes.feichtner@a-sit.at

Zusammenfassung: Die Verhaltensanalyse mobiler Applikationen für Apple iOS ist nach wie vor eine zeitlich und investigativ sehr herausfordernde Prozedur. Schlussendlich steht zumeist nicht klar fest, welche Maßnahmen eine Applikation vorsieht, um sensible Daten zu schützen. Gleichermaßen schwer ist es festzustellen, ob Apps gegen etablierte Sicherheitsprinzipien, etwa bei Verwendung kryptographischer Funktionen, verstoßen und so Angriffe auf kritische Daten begünstigen.

Im Rahmen dieses Projekts wurde eine Lösung gesucht, um iOS-Applikationen einer automatisierten statischen Analyse unterziehen zu können, die konkrete Aussagen über sicherheitsrelevante Eigenschaften liefern soll. Konkret sollte z.B. festgestellt werden können, ob Apps bei der Verwendung kryptographischer Funktionen Parameter einsetzen, die als hinreichend sicher gelten.

Da die statische Analyse von iOS-Anwendungen bislang ein vergleichsweise wenig erforschter Themenbereich ist, war im ersten Schritt dieses Projekts der Aufbau einer Wissensbasis über alle involvierten Komponenten vonnöten. Die wesentlichen Erkenntnisse daraus werden im Zuge dieses Berichts erläutert. Im zweiten Schritt wurde eine Analyse-Umgebung konzipiert, die eine vollautomatische Analyse von iOS-Anwendungen ermöglichen soll. Die Tauglichkeit der propagierten Lösung konnte im Rahmen einer Implementierung und dem Test mit realen Anwendungen eruiert werden. Neben Erkenntnissen über Schwächen der Umsetzung bei Apps, hat diese praktische Prüfung auch Einschränkungen der Lösung aufgezeigt, die einer schlüssigen Analyse hinderlich sein können.

Das vorgeschlagene Konzept sowie die praktische Umsetzung zeigen auf, dass die statische Analyse von iOS-Anwendungen in zielführender Weise durchführbar ist. Die Vielzahl an verfügbaren iOS-Anwendungen mit sicherheitsrelevanter Funktionalität drängen zudem die Notwendigkeit auf, künftig auch konkretere Untersuchungen einzelner Apps umzusetzen.

Dieses Dokument beschreibt die Ergebnisse des durchgeführten Projekts, stellt das erarbeitete Konzept vor und erläutert die Funktionsweise der prototypischen Umsetzung.

Inhaltsverzeichnis

1. Einleitung

Mit einem Marktanteil[1] von nahezu 12% ist Apple iOS nach Android das am häufigsten eingesetzte Betriebssystem für Mobilgeräte. Unter den 2 Millionen verfügbaren Anwendungen im „Apple App Store“ [2] finden sich zahlreiche Applikationen, die sicherheitsrelevante Funktionen implementieren, wie etwa Passwort-Manager oder Messenger-Dienste. Die Verhaltensanalyse mobiler Anwendungen für iOS ist nach wie vor ein zeitlich und investigativ sehr anspruchsvoller Vorgang. Während es zur Inspektion von Android-Apps bereits zahlreiche Werkzeuge und wissenschaftliche Beiträge gibt, sind diese Mittel aufgrund fundamentaler Plattformunterschiede nicht auf iOS übertragbar. Eine Überprüfung der sicherheitsrelevanten Eigenschaften von iOS-Apps erfordert somit neuartige Lösungsansätze bzw. eine auf die Plattform zugeschnittene Herangehensweise.

Anlässlich der großen Beliebtheit mobiler Plattformen werden auf entsprechenden Geräten auch Anwendungen ausgeführt, die sensible Daten verarbeiten. Vonseiten der Hersteller werden oftmals nur wenig Details dazu bekannt gegeben, auf welche Art und Weise heikle Informationen wie etwa Passwörter oder Schlüssel geschützt werden. Werden zu diesem Zweck kryptographische Funktionen eingesetzt, hängt das erreichbare Maß an Sicherheit in vielen Fällen von der Wahl geeigneter Parameter ab. Eine falsche Anwendung von Kryptographie oder sicherheitsrelevanten Schnittstellen kann dazu führen, dass der beabsichtigte Schutz signifikant geschwächt oder im schlimmsten Fall nicht mehr gewährleistet ist.

Im Zuge dieses Projekts wurde nach einem praxistauglichen Ansatz gesucht um Anwendungen analysieren zu können, ohne sie auf einem iOS-Gerät ausführen zu müssen. Dieser als „statische Analyse“ bekannte Ansatz zielt darauf ab, jeglichen in der App enthaltenen Code in die Inspektion einzuschließen. „Dynamische Analyse“, im Gegensatz dazu, wäre nur in der Lage, jene Teile der Anwendung zu betrachten, die aktuell ausgeführt werden. Beide Techniken stellen, abhängig vom Einsatzzweck und –ziel, geeignete Ansätze dar und liefern jeweils auch unterschiedliche Ergebnisse. Für die dynamische Analyse unter iOS sind mehrere Tools verfügbar[3], die das Verhalten von Anwendungen beobachten. Unter Verweis auf statische Analyse unter iOS finden sich zurzeit jedoch nur zwei wissenschaftliche Beiträge [1] [2], deren Hauptaugenmerk jeweils nicht auf den Einsatz von kryptographischen Funktionen in iOS-Apps gelegt wurde.

In den nachfolgenden Abschnitten dieses Dokuments wird erläutert, wie iOS-Anwendungen einer „statischen Analyse“ unterzogen werden können um Implementierungsschwächen zu erkennen. Als Einführung in die Thematik werden zunächst relevante Hintergründe zu iOS-Apps erklärt, gefolgt von einem Konzept für eine Analyseumgebung, die darauf ausgerichtet ist, sicherheitskritische Probleme in Anwendungen aufzuzeigen.

2. Grundlagen

Die Vielzahl der Anwendungen im App Store können im Hinblick auf ihre Komponenten in drei Kategorien unterteilt werden:

- **Nativ für die iOS-Plattform entwickelte Apps**
  Das Gros der verfügbaren Anwendungen wird in Objective-C[4] entwickelt, neuere Anwendungen verwenden bereits den designierten Nachfolger Swift[5]. Objective-C ist eine objektorientierte Programmiersprache, die auf C basiert und mit Ideen von Smalltalk erweitert wurde. Eine native Anwendung kann Code in Objective-C, C und C++ umfassen.

[1] https://www.idc.com/prodserv/smartphone-os-market-share.jsp - Daten zu 2016Q2.
[2] https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/
[3] https://www.owasp.org/index.php/IOS_Application_Security_Testing_Cheat_Sheet#Tools
[4] https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html
[5] https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html

- **Browser-basierte Apps**
  Unter browser-basierten Apps ist eine Art „mobilfreundlicher“ Klon einer Webanwendung zu verstehen. Derlei Applikationen basieren vollständig auf Webdateien (HTML, CSS, JavaScript), die über sog. WebViews vom System-Browser (MobileSafari) dargestellt werden. Wird eine browser-basierte Anwendung auf iOS gestartet, lädt die Startroutine den Browser und stellt die, entweder im App-Paket hinterlegte oder von einer Internetadresse abgerufene, Webseite dar. Eine Analyse dieser Kategorie von Apps muss somit auch die Sicherheitsprinzipien berücksichtigen, denen normale Webseiten entsprechen müssen[6]. Da browser-basierte Apps darüber hinaus keine, vom System bereitgestellten Schnittstellen (wie etwa Kryptographie) einsetzen können, spielen sie für die nunmehrige Analyse nur eine untergeordnete Rolle.

- **Hybride Anwendungen**
  Eine Mischung zwischen nativen und browser-basierten Applikationen werden als „hybrid“ bezeichnet. Typischerweise wird dieser Typ von Anwendung mit einem Wrapper ausgeliefert, der Webseiten über die WebView-Komponente darstellt. Darüber hinaus besteht jedoch die Möglichkeit, über geeignete Schnittstellen in native code, auch auf Systemkomponenten, etwa die Kamera oder das Mikrofon, zuzugreifen. Zumeist wird hierfür eine API in Objective-C bzw. Swift entwickelt, die eine Webanwendung dann über eine sog. „Bridge“ via JavaScript verwenden kann. Das zurzeit populärste Framework für hybride Anwendungen ist Apache Cordova[7]. Im Hinblick auf Sicherheitsaspekte vereint dieser Anwendungstyp die Anforderungen an beide zuvor angeführte Kategorien.

Im Unterschied zu Android, wo stack-basierter Java-Code zu register-basiertem Dalvik-Bytecode kompiliert wird, werden iOS-Anwendungen für die jeweiligen ARM CPU-Architekturen der Apple-Geräte zu native code kompiliert. Soll eine Anwendung beispielsweise auf einem iPhone 4S ausgeführt werden können, muss der komplette Programmcode für die ARMv7-Architektur kompiliert werden. Neuere Geräte (seit iPhone 5s) verwenden eine 64-bit fähige ARMv8[8]-CPU, die jedoch aus Kompatibilitätsgründen auch 32-bit Code für ARMv7s ausführen kann[9]. In der Praxis bedeutet das, dass die CPU eines iPhone 5s beispielsweise auch Apps ausführen kann, die für ältere iPhones kompiliert wurden.

Zusammenfassend ergibt sich somit eine *wesentliche Rahmenbedingung* für die statische Analyse von iOS-Apps: Für die Untersuchung einer Applikation muss eine Binärdatei, die Maschinencode für eine ARM-CPU enthält, in ein „lesbares“ Format gebracht werden. Ein, auch bei anderen Plattformen verbreiteter, Ansatz des „Reverse Engineering“ ist das Disassemblieren der Binärdatei zu Instruktionen in ARM-Assembler. Hierfür eignen sich beispielsweise die Tools Hopper[10] und IDA[11]. Angesichts der Komplexität und Größe heutiger Anwendungen erscheint eine Analyse auf dieser Ebene in der Praxis jedoch als fehleranfällig und nicht zielführend. Da Assembler-Code für jede Architektur unterschiedlich ist (z.B. gibt es bei ARMv7 Instruktionen, die es bei ARM64 nicht gibt), müsste eine Analyse-Umgebung für jede Architektur individuell konzipiert werden.

Als Abhilfe für die beschriebenen Einschränkungen wird somit vorgeschlagen, die Binärdatei in eine höhere Abstraktionsebene, vergleichbar mit Dalvik Bytecode unter Android, zu überführen. Der Ausgangspunkt hierfür ist naturgemäß weiterhin eine plattform-spezifische Implementierung, deren Abhängigkeiten jedoch im weiteren Verlauf wegfallen sollen, sodass auch Optimierungen des ursprünglichen Maschinencodes möglich werden.

---

[6] https://www.owasp.org/index.php/Web_Application_Security_Testing_Cheat_Sheet
[7] https://cordova.apache.org/
[8] Die Begriffe ARMv8, ARM64 und AArch64 werden fortan synonym verwendet.
[9] https://www.mikeash.com/pyblog/friday-qa-2013-09-27-arm64-and-you.html
[10] https://www.hopperapp.com/
[11] https://www.hex-rays.com/

2.1. Struktur von iOS-Apps

Als Format für Programme verwendet iOS (gleich wie das Desktop-Pendant OS X) „Mach object“-Dateien (Mach-O). Damit OS X und iOS-Anwendungen auf verschiedenen CPU-Architekturen ausgeführt werden können, werden sogenannte „Universal“ oder „Fat Binaries“ eingesetzt. Wird eine Anwendung ausgeführt, liest das Betriebssystem das Inhaltsverzeichnis der „Fat Binary“ (illustriert in Abbildung 2) und sucht darin wiederum nach einer Binärdatei für die aktuell verwendete CPU-Architektur. Eine „Fat Binary“ kann somit beispielsweise Mach-O-Binärdateien für iPhones mit ARMv7, neuere mit ARMv8-CPU und einen i386-Simulator umfassen.

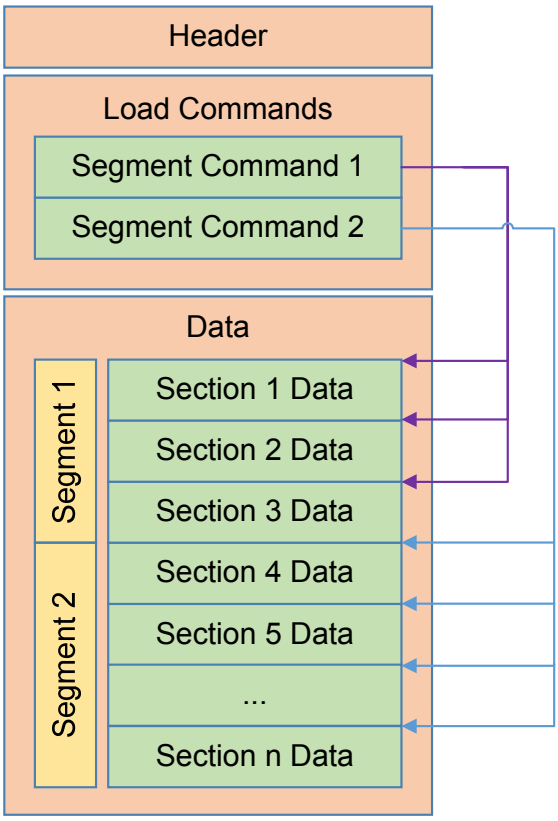

Abbildung 1. Mach-O-Binärdatei

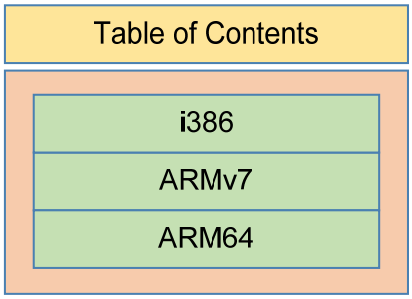

Abbildung 2. „Universal/Fat Binary“

Wie in Abbildung 1 dargestellt, bestehen Mach-O Dateien aus folgenden Komponenten[12]:

- **Header**
  Identifiziert das Dateiformat, beinhaltet genauere Angaben zur unterstützten CPU-Architektur und Informationen darüber, wie der Rest der Datei zu lesen ist.

- **Load Commands**
  Die auf den Header folgenden „Load Commands“ geben an, an welchen Positionen („Offsets“) der Datei sich die, einem Segment zugehörigen, einzelnen Bereiche der Binärdatei befinden. Im Segment „__TEXT“ wird z.B. auf die Position von ausführbaren Maschinencode verwiesen, auf konstante Strings oder die Position der Symbol table.

- **Data**
  Der letzte Bereich der Mach-O-Binärdatei enthält schließlich den ausführbaren Maschinencode und Daten, die bei Ausführung der Applikationen in den Arbeitsspeicher („Virtual Memory“) geladen werden. Ein sog. „Dynamic linker“ übernimmt schließlich die Auflösung von Referenzen auf Symbole der Objective-C Runtime und stellt letztlich sicher, dass alle Verweise („Pointer“) in der Binärdatei auf gültige Ziele verweisen.

[12] https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/MachOTopics/0-Introduction/introduction.html

2.2. Kompilieren einer iOS-Anwendung

Für die angestrebte Rückführung einer Mach-O-Binärdatei in ein leichter zu analysierendes Format, spielt es eine wesentliche Rolle, zu verstehen, wie eine Binärdatei zunächst gebaut wird. Abbildung 3 veranschaulicht den prinzipiellen Ablauf dieses Prozesses.

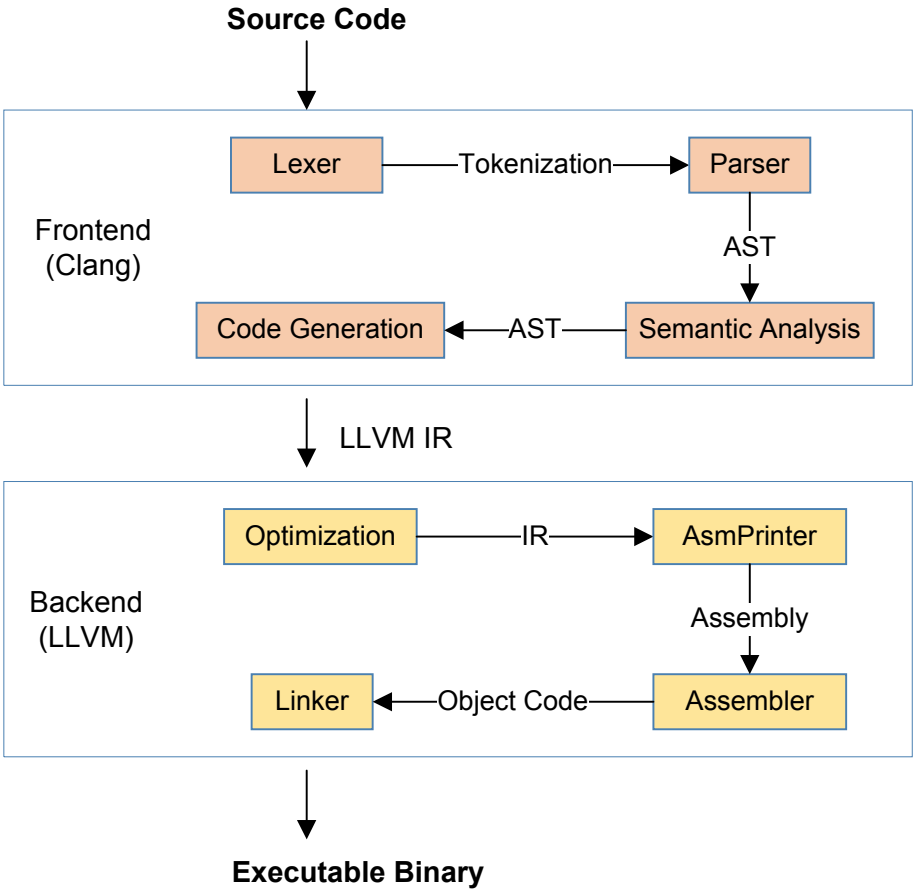

Abbildung 3. Schematischer Ablauf des Kompiliervorgangs.

Für die Übersetzung von Quellcode zu ausführbaren Binärdateien setzt Apple auf die Compiler-Infrastruktur LLVM[13]. Die Bereitstellung der entsprechenden Werkzeuge in LLVM geschieht durch das Frontend Clang[14].

Ausgehend von Quellcode in Objective-C, wird Quellcode im *Frontend* syntaktisch und semantisch geprüft. Der zuvor erstellte „Abstract Syntax Tree (AST)“ wird folglich in eine sog. LLVM Intermediate Representation[15] (LLVM IR) übertragen. Im *Backend* wird der Code zunächst allgemein optimiert (z.B. Entfernen von nicht aufgerufenem Code) und anschließend für eine gewisse CPU-Architektur übersetzt („AsmPrinter“-Schritt). Der „Assembler“-Schritt wandelt die Instruktionen in maschinenlesbaren Code um. Der Linker stellt abschließend sicher, dass Verweise auf externe Funktionen real existierende Ziele haben.

2.3. Fazit aus Grundlagen

Folgende Details aus dieser Beschreibung sind wesentlich für das im Weiteren vorgeschlagene Konzept einer Analyse-Umgebung:

- **Reduktion der Abhängigkeit von einer CPU-Architektur**
  Die statische Untersuchung einer iOS-Anwendung erfordert als Ausgangspunkt zunächst die

[13] http://llvm.org/docs/index.html
[14] https://www.objc.io/issues/6-build-tools/compiler/
[15] http://llvm.org/docs/LangRef.html

Festlegung auf eine, für eine gewisse CPU-Architektur kompilierte, Binärdatei. Der darauffolgend durchgeführte „Reverse Engineering"-Prozess muss auf die Charakteristika der gewählten Plattform eingehen. Da iOS-Geräte mit ARMv7-CPU in absehbarer Zeit verschwinden werden und der Instruktionssatz mit ARM64 im Vergleich zudem vereinfacht wurde, drängt sich die Wahl von ARM64 als Grundlage für die weitere Inspektion auf. Im allerersten Schritt der Analyse muss somit zuerst die ARM64-Binärdatei aus einer „Universal Binary" extrahiert werden.

Um den Inspektionsprozess in weiterer Folge von der Fixierung auf eine spezifische Architektur zu befreien, wird angestrebt, die Binärdatei zunächst zu LLVM IR-Code zurückzuführen. Obwohl diese Sprache hinsichtlich ihrer Struktur und Syntax Assembler nicht unähnlich ist, ist sie plattform-unabhängig und ermöglicht den Einsatz bereits bestehender statischer Analysetools, die mit LLVM IR-Code umgehen können. Konkret heißt das z.B., dass eine auf LLVM IR-Code ausgeführte Inspektionsroutine nicht speziell mit den Eigenheiten (wie der Calling Convention) von ARM64 vertraut sein muss.

- **Objective-C spezifische Eigenheiten im LLVM IR-Code**
  Angesichts des Optimierungsschritts, dem LLVM IR-Code beim Kompilieren unterzogen wird, gehen Informationen über mögliche Funktionsparameter und Rückgabewerte von Funktionen quasi absichtlich verloren. Da Objective-C eine sog. „runtime oriented language"[16] ist, werden so viele Entscheidungen wie möglich zur Laufzeit getroffen. Für die Analyse sehr relevant ist dieser Umstand beim Aufruf von Methoden. Die Entscheidung darüber, welche Methode aufgerufen werden soll, wird von der Objective-C Runtime Laufzeitumgebung getroffen. Eine Applikation teilt der Umgebung lediglich mit, welche Methode gestartet werden soll und ob sie statisch ist oder die Instanz einer Klasse benötigt. Diese Information wird der Laufzeitumgebung in Form einer „Message" übergeben. Beim Kompilieren werden direkte Methodenaufrufe somit transformiert zu „Messages" an die Laufzeitumgebung. Für das Verständnis des Kontrollflusses zwischen Funktionen ist es jedoch essentiell, beim „Reverse Engineering" diese Anpassungen wieder rückgängig zu machen und eine eindeutige Zuordnung von Methodenaufrufen zu Methoden zu schaffen.

- **Statische Analyse**
  Das Ziel der statischen Analyse ist, wie eingangs erwähnt, die Identifizierung von Implementierungsschwächen. In technischer Hinsicht gelingt das durch das Auffinden von Instruktionen und Variablen, die einen Einfluss auf einen wesentlichen Parameter eines sicherheitsrelevanten bzw. kryptographischen Funktionsaufruf haben.

  Für diese Feststellung ist es wesentlich, zu wissen, welche Werte von einem Programm an einer gewissen Stelle referenziert werden. Maschineninstruktionen greifen hierbei immer nur auf einen beschränkten, von der CPU bereitgestellten, Satz von Registern zurück, die in Instruktionen als Operanden fungieren. Oftmals verweisen diese Register auch auf gewisse Stellen im Speicher, wo Daten abgelegt sind. Dieses Schema ist dem Konzept von Pointer in anderen Programmiersprachen sehr ähnlich, wobei auch dort unterschiedliche Pointer-Variablen auf die gleiche Stelle im Speicher zeigen können. Die Kenntnis darüber, wohin Variablen bzw. in unserem Fall Register, zur Laufzeit hin verweisen könnten, ist eine wesentliche Notwendigkeit um alle Abhängigkeiten innerhalb der App auflösen zu können.

  Über diese, für eine aussagekräftige Analyse unumgängliche, „Pointer-Analyse" ist es schließlich möglich, herauszufinden, welche Instruktionen einer Anwendung relevant sind für einen gewissen Parameter-Wert, im einfachsten Fall z.B. all jene, die ihn direkt setzen. Es ermöglicht gleichermaßen, durch programmatisches Verständnis des Aufbaus von Mach-O-Binärdateien, zu ermitteln, wo sich die Klassen befinden, auf die Methodenaufrufe verweisen.

  Zusätzlich zur „Pointer-Analyse" gilt es, den Informationsfluss festzustellen, der an einer gewissen Variable endet. Dies bedeutet, Sequenzen von Instruktionen zu ermitteln, die einen

[16] http://cocoasamurai.blogspot.co.at/2010/01/understanding-objective-c-runtime.html

Einfluss auf die den endgültigen Wert eines Parameters haben, wie z.B. den bei einer AES-Verschlüsselung verwendeten Schlüssel. Das hierfür anzuwendende Verfahren ist als „Program Slicing“ [3] bekannt.

3. Konzept einer Analyse-Umgebung

Basierend auf den zuvor eruierten Rahmenbedingungen, wird nachfolgend eine Architektur vorgeschlagen, die dazu dienen soll, iOS-Anwendungen auf vorgegebene Implementierungsschwächen hin zu untersuchen.

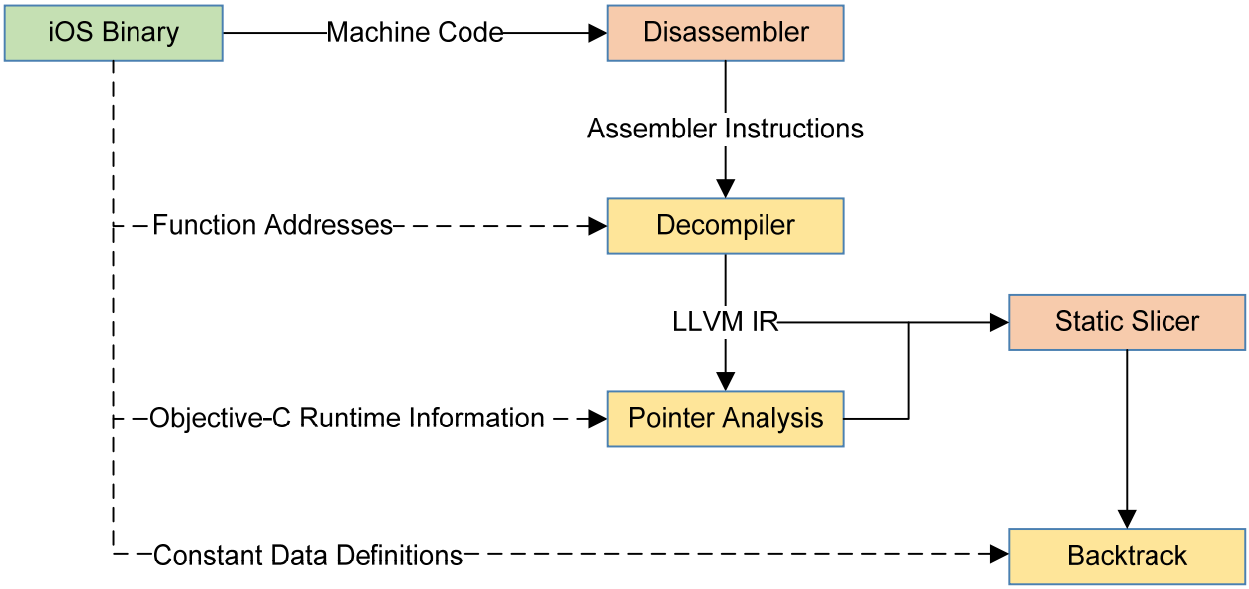

Abbildung 4. Ablaufdiagramm der statischen Analyse einer iOS-Anwendung.

Abbildung 4 veranschaulicht die wesentlichen Schritte, die als notwendig erachtet werden, um iOS-Binärdateien zu untersuchen. Die einzelnen Schritte sind wie folgt:

(0. Extrahieren der ARM64-Binärdatei aus einer „Universal Binary“)

1. **Disassembler:** Im ersten Schritt muss der maschinenlesbare Code als ARM64-Assemblercode dargestellt werden.
2. **Decompiler:** Die Aufgabe des „Decompiler“ ist es, die Schritte, die beim Kompilierungsprozess im Backend von LLVM übernommen wurde, soweit als möglich zurück zu übersetzen. Die Gabe von ARM64-Assemblercode soll somit zu LLVM IR-Code führen. Um Verweise auf Funktionsaufrufe aufzulösen, muss der Decompiler die entsprechenden Informationen aus den Segmenten der Mach-O-Binärdatei beziehen.
3. **Pointer Analysis:** Als eine Art Unterstützung für die weitere Analyse dient die „Pointer-Analyse“ zur Prüfung, auf welche Speicherstellen ein Zeiger hin verweist. Pointer nicht aufzulösen, würde beim nachfolgenden „Slicing“ zu falschen Ergebnissen führen.
4. **Static Slicing:** Über „Slicing“ lassen sich relevante Codeteile im Programm anhand eines sog. „Slicing Criterion“ identifizieren. Ist das „Slicing Criterion“ beispielsweise ein sicherheitskritischer Parameter eines kryptographischen Funktionsaufrufs, lässt sich über Slicing dessen Ursprung ermitteln.
5. **Backtrack:** Beim „Slicing“ ergeben sich Pfade, die aufzeigen, welche Schritte bzw. Instruktion einen Parameter beeinflussen. Der Backtracking-Schritt analysiert diese Pfade schließlich und inspiziert, ob ein Wert letztlich aus einer „sicheren Quelle“ kommt, konstant ist oder einen anderen Ursprung hat. Ausgehend darauf, lässt sich feststellen, ob eine Implementierungsschwäche vorliegt oder nicht.

4. Prototypische Implementierung

Die Realisierung des vorgelegten Ablaufdiagramms erfordert in allen Punkten neue bzw. stark angepasste Komponenten existierender Implementierungen. Da die Compiler-Infrastruktur LLVM

bereits einen Disassembler integriert, bietet es sich an, diesen für die Rückwandlung von Maschinencode zu ARM64-Assembler[17] einzusetzen.

4.1. Decompiler

Um ARM64-Assemblercode zu LLVM IR-Code zu übersetzen, bedarf es eines Decompilers, der den Kontroll- und Datenfluss des Programms korrekt repliziert. Um direkt am disassemblierten Code aufsetzen zu können, wurde nach einem Decompiler gesucht, der auf LLVM basiert bzw. sich als LLVM-Modul integrieren lässt.

Ein Test, der zur Auswahl stehenden Frameworks Dagger[18][19], Fracture[20], libbeauty[21] und McSema[22], ließ die Auswahl auf Dagger fallen. Dagger verwendet bereits bestehende Backends bei LLVM und deren Beschreibung von Registern und Instruktionen. Dadurch lässt sich die Semantik einzelner Instruktionen in Maschinencode im entsprechenden LLVM IR-Code darstellen. Diese Übersetzung ist relativ „simpel“, da Dagger jene Semantik-Definitionen dafür verwenden kann, die LLVM normalerweise in Vorwärtsrichtung, also bei der Umwandlung von LLVM IR-Code zu ARM64-Assemblercode, einsetzt.

Leider sind diese Definitionen nicht für alle Instruktionen vorhanden. Um den Prozess dennoch durchführen zu können, muss Dagger daher manuell um fehlende Definitionen ergänzt werden. Im Endeffekt muss somit jede Instruktion ein Pendant als LLVM IR-Instruktionen aufweisen.

4.2. Pointer-Analyse

Zur Berechnung auf welche Speicherstellen ein Pointer verweist, wurde nach einer Lösung gesucht, diese Analyse auf Basis von LLVM IR-Code vornehmen kann. Es fand sich hierfür die Implementierung[23] eines Algorithmus von Andersen [4], der in der vorliegenden Form direkt übernommen werden kann

4.3. Static Slicer

Die vom Slicer vorgenommene Datenflussanalyse wird direkt auf LLVM IR-Code betrieben. Basierend auf einem sog. „Slicing Criterion“ (Startbedingung), werden alle Pfade zurückverfolgt, die einen Einfluss auf das Criterion haben.

Zu diesem Zweck wurde eine Implementierung[24] aufgefunden, die das Konzept des „Slicing“ bereits implementiert und sich auf LLVM IR-Code anwenden lässt. Obwohl dort sogar schon eine Pointer Analyse enthalten ist, scheint es sinnvoller, die Analyse von Andersen einzusetzen. Ein Hauptgrund dafür ist, dass die Andersen-Methode effizienter mit größeren Binärdateien, wie eben iOS-Applikationen, arbeitet. Eine Integration ist relativ einfach, da die Pointer-Analyse nichts mit eigentlichen Logik des „Slicing“ zu tun hat.

4.4. Backtracking

Nach der Ausführung des Slicers ergeben sich alle logischen Ausführungspfade, die den Wert des zuvor spezifizierten Criterion an einem gewissen Punkt beeinflussen können. Um letztlich zu prüfen, ob eine Implementierungsschwäche vorliegt, müssen diese Pfade durch Backtracking untersucht werden. Praktisch bedeutet das, jene Instruktionen zu identifizieren, wo einer Variable tatsächlich ein Wert, und vor allem welcher, zugewiesen wird. Während Slicing somit alle Statements im Slice inkludiert, die irgendwie im Zusammenhang mit dem Criterion stehen, zielt Backtracking gezielt auf einzelne Variablen ab.

---

[17] https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf
[18] http://dagger.repzret.org/
[19] https://github.com/repzret/dagger
[20] https://github.com/draperlaboratory/fracture
[21] https://github.com/jcdutton/libbeauty
[22] https://github.com/trailofbits/mcsema
[23] https://github.com/grievejia/andersen
[24] https://github.com/jirislaby/LLVMSlicer

In der praktischen Umsetzung gelingt diese Auswertung durch das Verfolgen der Pfade bis hin zu ihren Endpunkten. Das Problem ist vergleichbar mit der Suche in einem Graphen und im Grunde relativ trivial durchführbar, wenn dabei sichergestellt wird, dass Wege nicht mehrmals beschritten werden (Möglichkeit von Endlosschleifen).

5. Analyse von iOS-Apps in der Praxis

Um die Tauglichkeit der Analyse-Umgebung zu testen, wurden eine Vielzahl an iOS-Anwendungen aus dem Apple App Store heruntergeladen. Da für keine dieser Anwendungen der Quelltext der Applikation verfügbar ist, lassen sich die Ergebnisse der Inspektion allerdings nur schwer nachprüfen. Um die Tauglichkeit der Analyse-Umgebung sicherzustellen, wurde daher mit einer quelloffenen App namens „Damn Vulnerable iOS App“[25] (DVIA) getestet. Der Vorteil dieser speziellen Anwendung ist, dass sie prinzipiell dafür konzipiert ist, möglichst viele sicherheitskritische Implementierungsprobleme zu beinhalten. Dazu gehören auch Unzulänglichkeiten bei der Verwendung von kryptographischen Funktionen.

Folgende Probleme ließen sich in dieser Anwendung automatisch feststellen und manuell verifizieren:

- **Verwendung des ECB-Modus für symmetrische Blockchiffren**
  Die Erkennung dieses Problems funktioniert sehr trivial, da dieser Modus erst durch Setzen einer gewissen Konstante aktiviert wird. Ist sie nicht gesetzt, verwendet iOS standardmäßig[26] Verschlüsselung im sichereren CBC-Modus.

- **Statischer Initialisierungsvektor (IV) für Verschlüsselung**
  Ein konstanter bzw. statischer IV wird immer durch ein Array von Bytes definiert. Wird dieses Array dynamisch (also zur Laufzeit) erst zusammengesetzt, z.B. durch Einfügen einzelner Elemente, ist eine Erkennung zurzeit leider nicht möglich. Die Schwierigkeit dabei liegt vor allem in der Feststellung, dass es sich de facto um einen konstanten Wert handelt, obwohl er dynamisch erst als solcher zusammengesetzt wird.

- **Zu wenig Iterationen bei der Schlüsselableitung mittels PBKDF**
  Zur Erkennung wie viele Durchläufe beim Aufruf der Funktion *CCKeyDerivationPBKDF()* durchgeführt werden, ist die einfache Auswertung eines Integer-Werts vonnöten. Dies funktioniert insofern relativ gut, da ein Zahlenwert sehr eindeutig zu interpretieren ist.

- **Konstanter Salt für Schlüsselableitung mittels PBKDF**
  Hierbei wird geprüft, ob der Wert, der der Funktion SecRandomCopyBytes() übergeben wird, möglicherweise nicht zufallsgeneriert ist. Die Auswertung des Backtracking funktioniert ähnlich wie beim statischen IV, weswegen auch die gleichen Einschränkungen zum Tragen kommen.

- **Konstanter Schlüssel zur Verschlüsselung** *und*
  Konstantes Passwort für PBKDF
  Die Erkennung dieser beiden Missstände gestaltet sich nicht immer einfach, wenn es viele Pfade gibt. Wenngleich es bei der DVIA-App nur 2 Pfade gab, die beide auf einen konstanten Wert verwiesen, fand sich unter den heruntergeladenen Apps auch eine mit 7000 Pfaden. Die Schwierigkeit in diesem Fall ist jedoch wohlgemerkt nicht die Anzahl der möglichen Ausführungspfade, sondern die dadurch entstehende mangelnde Aussagekraft. Gibt es dabei sowohl mit konstanten Endpunkten, als auch mit dynamisch konstruierten, d.h. ein verwendeter Schlüssel bzw. Passwort erscheint potentiell konstant als auch variabel, kann keine schlüssige Aussage getroffen werden.

Bei der Inspektion von beliebigen Apps aus dem App Store, offenbarte sich, dass viele Apps bei der Wahl geeigneter Parameter für kryptographische Funktionen gleichen Fehler begehen:

[25] https://github.com/prateek147/DVIA
[26] Hinweis: Bei Android verhält sich dieser Umstand umgekehrt. Standardmäßig wird ECB-Modus eingesetzt.

- **Verwendung eines Passworts als Schlüssel für AES**
  Da unter iOS sowohl ein Byte Array wie ein String durch einen Pointer referenziert der Funktion *CCCrypt(..)* als kryptographischer Schlüssel übergeben werden können, wird offenbar des Öfteren fälschlicherweise versucht, ein Passwort als Schlüssel zu verwenden. Ohne eine vorherige Ableitung zu einem sicheren kryptographischen Schlüssel, schwächt ein Passwort beliebiger Länge, die Wirksamkeit der Verschlüsselung signifikant.

- **Verwendung von NULL als Initialization Vector**
  Wird als IV der Funktion *CCCrypt(...)* ein Pointer mit dem Wert NULL übergeben, wird streng genommen ebenfalls ein konstanter Wert verwendet. Der Einsatz von NULL führt dabei zu ähnlichen Implikationen wie die Verwendung eines konstanten Arrays.

5.1. Einschränkungen

Wenngleich die implementierte Analyse-Umgebung die Praxistauglichkeit des Konzepts unterstreicht, gibt es selbstverständlich viele relevante Aspekte, die für einen produktiven Einsatz der Lösung beachtet werden müssten. Essentiell erscheinen nach Durchführung dieses Projekts folgende Erkenntnisse:

- Die Funktionsfähigkeit des Decompiler hängt davon ab, ob jede, von einer App verwendeten, Instruktion ein Pendant in LLVM IR-Code hat. Beim Versuch mit beliebigen Apps aus dem App Store hat sich gezeigt, dass hier Nachbesserungsbedarf besteht.

- Die derzeitige Implementierung hat die Charakteristika von Swift außen vorgelassen. Der größte Unterschied zwischen Objective-C und Swift bei der Analyse ist der Umstand wie Methoden aufgerufen werden. Während dies bei Objective-C über „Messages“ an die Runtime geschieht, verwendet Swift eine Methodentabelle wie bei C/C++.

- Die Analyse findet von der Konzeption her nur jene Schwächen in Apps, nach denen a priori gesucht wird. Die DVIA-App beispielsweise beinhaltet eine Unmenge weiterer Probleme, die aktuell nicht beachtet werden. Eine Erweiterung der Umgebung um auch andere Probleme zu finden, ist aber prinzipiell möglich und sinnvoll.

6. Fazit

Im Zuge dieses Projekts wurde ein Konzept erarbeitet und praktisch umgesetzt, anhand dessen Anwendungen für das Mobilbetriebssystem iOS analysiert werden können. Es ist dadurch möglich, Implementierungsschwächen aufzudecken, die in Relation zum Aufruf kryptographischer Funktionen stehen und durch die Wahl schlechter oder geeigneter Parameter bedingt sind.

Angesichts der Einschränkung, dass bei iOS-Anwendungen direkt die Binärdatei analysiert werden muss und ohne Weiteres keine dienlichere Repräsentation zur Verfügung steht, gibt es bislang nur sehr wenig Arbeiten, die ähnliche Absichten bestreiten. In diesem Projekt wurde eine praxistaugliche Lösung vorgestellt, um beliebige Anwendungen zunächst in eine analysierbare Sprache übersetzen zu lassen um sie dann mit etablierten Werkzeugen zu untersuchen. Durch die Entkoppelung der Ausgangssprache hin zu einer abstrakteren Darstellung schafft auch eine Basis für eine mögliche Erweiterung oder anderweitige Analyse einer Applikation.

Die Analyse von Apps in der Praxis hat aufgezeigt, dass die Analyse-Umgebung in der Lage ist, die gesuchten Probleme in Apps auch tatsächlich zu identifizieren. Dies untermauert zum einen die Tauglichkeit des Konzepts und seiner Implementierung. Zum anderen streicht es die Tatsache hervor, dass es eine aktive Notwendigkeit gibt, Analysen von Apps, wie sie bisher nur mit Android durchgeführt wurden, auch auf iOS umzusetzen.

7. Literaturverzeichnis

[1] Z. Deng, B. Saltaformaggio, X. Zhang und D. Xu, „iRiS: Vetting Private API Abuse in iOS Applications," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS),* 2015.

[2] M. Egele, C. Kruegel, E. Kirda und G. Vigna, „PiOS: Detecting Privacy Leaks in iOS Applications," *Proceedings of the 18th Annual Network & Distributed System Security Symposium (NDSS),* 2011.

[3] M. Weiser, „Program Slicing," *IEEE Transactions on Software Engineering SE-10.4,* 1984.

[4] L. O. Andersen, „Program analysis and specialization for the C programming language," *Writing May,* 1994.