Obfuskation von Quellcode in mobilen und webbasierten Anwendungen

Wilhelm Büchner Hochschule

Alexander Pann

850621

10. Mai 2021

Inhaltsverzeichnis

1 Einführung

Laut Purplesec [44] ist die Cyberkriminalität während der COVID-19-Pandemie um 600 % gestiegen. Unter den Betroffenen sind bei Webanwendungen Unternehmen wie beispielsweise Twitter, bei der ein gut koordinierter Betrug dazu führte, dass Angreifer durch fast 300 Transaktionen 121000 US-Dollar in Bitcoin erbeuteten. Auch das Unternehmen Zoom war betroffen, bei dem 500000 Passwörter gestohlen wurden, die im Dark Web zum Kauf angeboten wurden. Jedes Jahr werden die OWASP Top Ten der Sicherheitsrisiken von Webanwendungen [49] veröffentlicht und auch Handbücher für Sicherheit von Android- und iOS-Anwendungen von den jeweiligen Unternehmen verfasst [16] [2]. Diese Arbeit beschäftigt sich mit dem Thema Obfuskation, wobei ein Quellcode so verändert wird, dass er für Menschen schwerer verständlich und zurückgewinnbar (Reverse Engineering) wird. Es wird dabei das Reverse Engineering des originallen Codes erschwert.

1.1 Definition

Laut Merriam-Webster [34] kommt das Wort Obfuskation aus dem Lateinischen von ob + fuscare und bedeutet verdunkeln. In der Softwareindustrie bezieht es sich auf eine Technik, bei der Code verändert wird, wobei das Verhalten beibehalten, die Struktur und Absicht jedoch verschleiert wird. In dieser Arbeit wird Verschleierung als Synonym für Obfuskation verwendet.

1.2 Abgrenzung

Miessler [35] grenzt dieses Technik zu verwandten Themengebieten ab:

Encodierung
: Encodierung dient der Aufrechterhaltung der Nutzbarkeit der Daten und kann mit demselben Algorithmus rückgängig gemacht werden, mit dem der Inhalt verschlüsselt wurde, d. h. es wird kein Schlüssel verwendet.

Verschlüsselung
: Die Verschlüsselung dient der Aufrechterhaltung der Vertraulichkeit der Daten und erfordert die Verwendung eines geheim gehaltenen Schlüssels, um den Klartext zurückzugewinnen.

Hashing
: Hashing dient zur Überprüfung der Integrität von Inhalten, indem alle Modifikationen durch offensichtliche Änderungen an der Ausgabe des Hashwertes erkannt werden.

Obfuskation

Verschleierung verhindert, dass Menschen die Lesbarkeit von etwas für Menschen erschwert wird und wird oft bei Quellcodes angewandt, um erfolgreiches Reverse Engineering und/oder Diebstahl der Funktionalität eines Produkts zu verhindern.

Obfuskation erschwert das Verständnis daher nur für den Menschen und kann leicht von Computern verarbeitet werden. Bei der Verschlüsselung ist das Lesen ohne Schlüssel weder für Mensch noch Maschine möglich. Diese Technik bietet daher keinen starken Schutz vor Angreifern im Gegensatz zu einer richtig eingesetzten Verschlüsselung, sondern bietet eher nur ein Hindernis. Obfuskierter Code kann oft wie bei einer Verschlüsselung bis zu einem gewissen Grad mit der gleichen Technik rückgängig gemacht werden, mit der sie verschleiert wurde. Ein weiterer kritische Punkt bei Verschleierung ist, dass es eine Begrenzung dafür gibt, wie verschleiert der Code werden kann. Dies ist abhängig von dem Inhalt, der verschleiert wird. Bei Quellcode besteht die Einschränkung darin, dass das Ergebnis immer noch vom Computer verarbeitet werden können muss, da die Anwendung sonst nicht mehr funktioniert. Sowohl Obfuskation als auch Verschlüsselung zählen laut Collberg et al. [8] zum technischen Schutz von intellektuellem Eigentum (in diesem Fall: Quellcode).

2 Techniken der Code-Obfuskation

Die Autoren beschreiben auch verschiedene Techniken zur Transformation von Code, um Softwaregeheimnisse zu schützen. Die Obfuskation basiert dabei auf Code-Transformationen, die ähnlich wie Optimierungen des Kompiliers funktionieren. Der Begriff *Methode* bezieht sich in den folgenden Absätzen auf Funktionen/Prozeduren der objektorientierten Programmierung. Aggegration-Transformationen trennen logisch zusammengehören Berechnungen auf oder führen Berechnungen zusammen, die logisch nicht zusammengehören. Die einfachste Transformation ist die einseitige Transformation des Layouts, bei der die spezifische Formatierung des Quellcodes in der jeweiligen Programmiersprache entfernt wird. Das Mischen von Bezeichnern ist auch eine Einwegfunktion, die allerdings effektiver ist. Die nachfolgende Liste beinhaltet rechnerische Umformungen, bei denen der Kontrollfluss des Programms verschleiert wird:

Einfügung von toten oder irrelevanten Code erhöht die wahrgenommene Komplexität des Codes.

Erweiterung von Schleifenbedingungen durch das Verkomplizieren der Abbruchbedingung ohne die Programmlogik zu verändern.

Änderung eines reduzierbaren in einen nicht reduz. Flussgraphen
Programmiersprachen werden oft in nativen Code oder Code für eine virtuelle Maschine kompiliert. Dabei können Transformationen verwendet werden, bei der ein Maschinencode oder nativer Code keine äquivalenten Befehle in der Quellsprache besitzt.

Entfernung von Bibliotheksaufrufen und Mustern wie z. B. die Ersetzung einer bekannten Datenstruktur durch eine weniger bekannte Struktur.

Interpretation mit Tabellen bei der ein Teil des Quellcodes in einen anderen virtuellen Maschinencode umgewandelt wird. Dieser wird von einem Interpreter für virtuelle Maschinen ausgeführt, der in der verschleierten Applikation eingefügt wurde.

Hinzufügung von redundanten Operanden wie z.B. der Multiplikation mit 1 bei Ganzzahloperationen.

Parallelisierung von Code durch die Erstellung von Scheinprozessen ohne sinnvolle Aufgabe oder das Aufteilen eines sequenziellen Abschnitts in mehrere Abschnitte, die parallel ausgeführt werden.

Aggregation
In diesem Abschnitt werden einige Aggregations-Techniken vorgestellt. Beim Eingliedern von Methoden wird der Inhalt der Methode direkt an die ausführende Stelle kopiert. Dies ist eine bekannte Optimierungstechnik bei Kompilern. Beim Ausgliederung wird eine Sequenz von Anweisungen zu einer Unterroutine zusammengefügt. Durch das Verschachteln von Code kann das Reverse Engineering laut Rugaber et al. [47] deutlich erschwert werden. Durch das Erzeugen von Methoden mit ähnlichen Signaturen kann der tatsächliche Methodenaufruf verschleiert werden. Optimierungen von Schleifen lässt zusätzlich Komplexität entstehen. Bekannte Techniken sind Schleifenabrollungen [24] und Schleifenspaltungen [53]. Durch die Randomisierung der Reihenfolge von Elementen im Quellcode wird auch das Verständnis des Codes erschwert, da Programmierer ihren Quellcode meist so organisieren, dass seine Lokalität maximiert wird.

Die folgende Liste beschreibt Transformationen, bei der die Datenstruktur verschleiert wird:

Änderungen der Speicherung und Kodierung

Bei Speicher-Transformationen werden unnatürliche Speicherklassen für statische und dynamische Daten gewählt. Bei Kodierungs-Transformationen werden unnatürliche Kodierungen für häufige Datentypen verwendet. Die beiden Techniken können gemeinsam oder getrennt voneinander verwenden werden. Eine Zahl i kann beispielsweise durch eine einfache Kodierungsfunktion $i' = c_1 \cdot i + c_2$ verändert werden. Variablen können auch von einer spezialisierten Speicherklasse zu einer allgemeineren Klasse aufsteigen (z.B. von einer Ganzzahlvariable zu einem Ganzzahlobjekt in Java). Auch die Lebensdauer einer Variable kann durch Ändern von Attributen verändert werden (z.B: durch das Ändern einer lokalen Variable zu einer globalen Variable). Boolesche Variablen und andere Variablen mit eingeschränkten Bereich können auf mehrere Variablen aufgeteilt werden. Statische Daten können in prozedurale Daten verändert werden. Im Falle einer Zeichenkette würde dies beispielsweise bedeuten, dass nach der Transformation ein Programm die Zeichenkette erzeugt und die Zeichenkette nicht mehr statisch in das Programm eingebunden wird.

Aggregation

In diesem Abschnitt werden erneut einige Aggregations-Techniken vorgestellt. Zwei oder mehrere skalare Variablen $V_1..V_K$ können zu einer Variable V_M kombiniert werden. Zwei 32-Bit Ganzzahlen können beispielsweise zu einer 64-Bit-Variable kombiniert werden. Verschiedene Transformationen von Feldern führen zu einer besseren Verschleierung: Sie können in mehrere Unterfelder aufgeteilt werden oder mehrere Felder können zu einem Feld zusammengeführt werden. Durch eine Falt-Operation kann die Dimensionalität des Feldes erhöht werden. Durch eine Verflachung wird eine Reduzierung der Dimensionalität erreicht. In objektorientierten Sprachen kann auch die Vererbungsbeziehung verändert werden. Eine ähnliche Technik ist das falsche Refactoring, das bei zwei Klassen A und B angewandt wird, die keine gemeinsamen Verhalten besitzen. Wenn beide Klassen Instanzvariablen desselben Typs besitzen, können diese in die neue Elternklasse C verschoben werden. Die Methoden von C können fehlerhafte Versionen der Methoden aus A und B sein.

Änderung der Reihenfolge von Methoden und Instanzvariablen innerhalb von Klassen sowie formalen Parametern innerhalb Methoden.

3 Plattformabhängige Verschleierung

Verschleierungstechniken sind nicht nur von Programmiersprache zu Programmiersprache abhängig, sondern auch von der verwendeten Plattform. Laut statista.com [48] waren im Oktober 2020 mit 72.92% Android und mit 26.53% iOS die führenden mobilen Betriebssysteme, die zusammen fast 99% des gesamten Marktes abdecken. Daher werden in den nächsten Abschnitten die beiden mobilen Plattformen Android und iOS betrachtet sowie Webapplikationen im Allgemeinen, da Webseiten oft angegriffen werden.

3.1 Web-Apps

In diesem Kapitel werden häufige Techniken der Obfuskation in Webanwendungen vorgestellt. Aufgrund der Vielzahl an Werkzeugen wird hier stattdessen auf Techniken der Webtechnologien HTML und JavaScript eingegangen (vgl. Heiderich et al. [22]).

3.1.1 HTML

HTML kann verschleiert werden, um Eingabefilter von Webanwendungen zu umgehen oder auch um forensische Arbeiten zu verlangsamen. Dabei wird oft der Parser ausgenutzt. Die folgende Liste demonstriert einige Techniken, die teilweise nur bei speziellen Browserversionen funktionieren:

Verschleierte Tagnamen z.B: $<L\mu\, onclick = alert(1)>clickme</L\mu>$

Verschleierte Trennzeichen z.B: $<img\backslash 0/\, src = x\, onerror = alert(1)//>$

Mehrere gleichnamige Attribute
z.B: $<span\, style = \text{“}color : red\text{“}\, style = \text{“}color : green\text{“}\, />$

Tücken von schließenden Tags z.B:
$</p<img\, src = x\, onerror = alert(1)>$
$</br<img\, src = x\, onerror = alert(2)>$

Einbindung von Scripts in Tags z.B:
$<object\, data = \text{“}javascript : alert(1)\text{“}>$

Bedingte Kommentare z.B:
$<! - -[if\, IE\, 8]><p>Welcome\, to\, Internet\, Explorer\, 8.</p><![endif] - ->$

JavaScript URIs z.B:
$<a\, href = \text{“}j\&\#x61vascript : \%61lert(1)\text{“}>clickme$

Defekte Protokoll-Handler z.B:
$<a\,href = \text{“}j\&\#x61vascript:\ //\%0\&\#x61\,\&\#x00025;61lert(1)\text{“}>a$

Data-URIs z.B:
$<iframe\,src = \text{“}data:\mu,<script>alert(document.domain)</script>\text{“}>$

Event-Handler z.B:
$<body\,onload = \text{“}al\&\#000101rt\&\#8233//*\,\&\#00*/(1)//\text{“}$

3.1.2 JavaScript

Auch JavaScript kann verschleiert werden. Die folgende Liste demonstriert einige Techniken, die in den meisten Browsern funktioniert sollten:

Mehrzeilige Zeichenketten z.B:
$alert(\text{“}this\,is\,a\ \backslash$
$string\text{“})$

Reguläre Ausdrücke als Funktionen z.B: $alert(/a(a)(b)jc/g(\prime aab\prime));$

Unicode-Escapezeichen z.B: $alert(1);\backslash u0061ler\ \backslash\ u0074(1);$

Hexadecimal-Escapezeichen z.B: $eval(\prime\backslash x61lert(1)\prime);$

Oktal-Escapezeichen z.B: $eval(\prime\backslash 141lert(1)\prime);$

Vom Benutzer definierte Variablen dürfen Zahlen (außer am Anfang), Unterstriche (_), Dollarzeichen und sämtliche Unicodezeichen enthalten. Bei einer beliebten Webobfuskationtechnik werden Variablen erstellt, die keine alphanumerischen Zeichen enthalten. Ein Beispiel dazu kann in Abbildung 1 betrachtet werden.

Zu Verschleierungszwecken werden auch oft die folgenden eingebauten Variablen verwendet:

- window
- window.name
- location.hash
- document.URL

3.2 Android

Android-Apps werden in Java geschrieben und wurden früher von der Dalvik Virtual Machine [11] ausgeführt. Seit Android 5 wird die Android Runtime (Art) [17] verwendet. Der Java-Code wird zu Bytecode kompiliert und von der

```
_=~[];_={_:{_:!![]+[],$_:![]+[],$:[]+{},__:[][_]+[]},_$:++_,$_:++
_,$$:-~_++,$:-~_,__:-~++_},_._.$$_=_._.$[_.__+_.$_]+_._.$[_.$_]+
_._.__[_.$_]+_._.$_[_.$]+_._._[_._$]+_._._[_.$_]+_._.__[_._$]+_._.$[
_.__+_.$_]+_._._[_._$]+_._.$[_.$_]+_._._[_.$_],_.$$_=[][_._.$$_][
_._.$$_],_._.___=[]+/[]/[_._.$$_],_._.$$=[]+([]+[])[_._.$$_],_._.__$
=_._._[_._$]+_._.$[_.$_]+_._.$$[_.$_+_.__+_.__]+_._._[_._$]+_._._[_.$
_]+_._.__[_.$_+_.__]+_._.__[_.$_]+_._.$$[[]+_.$_+_.__];_._.$__=_._._[
_._.$_[_.$$]+_._.__[_.$_+_.__]+_._.__[_.$_]+((_.$$+_.$)*_.__)[
_._.__$](_.$*(_.$+_.__))]()[_._.$$$=_._._[_.$_]+_._._[_.$]+_._._[
_._$]+_._._[_.$$]+_._._[_.$_]+_._.__[_.$_];_._._$$=[]+_.$$_(_._.$$$+
_._.$[_.$+_.__]+([]+~_._$)[_._$]+_.$_+_._.$__[[]+_.$$+(_.$$+_.$)]+
_.$$)(),_.$$$=_.$$_(_._.$$$+_._.$[_.$+_.__]+_._._[_.$]+_._.$_[_.$]+
_._.$[_.__+_.$_]+_._.$_[_.$_]+_._.___[[]+_.$_+_.__]+_._._[_.$])(),_.$
__=_.$$_(_._.$$$+_._.$[_.$+_.__]+_._._[_.$$]+_._.__[_.$_]+_._._[_.$]+
_._.$_[_.$]+_._.$[_.__+_.$_]+_._.$_[_.$_]+_._.___[[]+_.$_+_.__]+_._._[
_.$])();_._._$=[]+_.$$$(_._.$[_.$+_.__])[_._$];$={$:_._.$_[_.$_],__:
_._.$_[_.$$],$_:_._._[_.$],$$:_._._[_.$_],_$:_._._[_._$],___:_._.$$[
_.$*(_.$$+_.$)],_:_.$_,$__:_._.$$[_.__*_.__]};_.$$_($.$+$.__+$.$_+$
.$$+$._$+$.___+$._+$.$__)()
```

Abbildung 1: JavaScript-Code für: alert(1)

virtuellen Maschine ausgeführt. Es können auch andere Programmiersprachen und Tools zur Entwicklung verwendet werden, auf die hier nicht näher eingegangen wird. Zum Schutz vom Bytecode können verschiedene Tools verwendet werden:

ProGuard [20] ist ein Open Source-Tool zum Verbessern, Verschleiern und Optimieren von Java-Bytecode. Es wurde lange Zeit im Android Studio zur Obfuskation von Androidanwendungen verwendet. Es schützt vor statischer Analyse.

DexGuard [19] ist ein spezialisiertes Werkzeug für den Schutz von Android-Anwendungen. Im Gegensatz zu ProGuard schützt es Anwendungen gegen statische als auch dynamische Analyse. Es wendet dazu mehre Schichten von Verschlüsselung und Obfuskation an. Neben dem Dalvik-Bytecode werden Manifestdateien, native Bibliotheken, Ressourcen, Ressourcendateien und Asset-Dateien optimiert, obfuskiert und verschlüsselt. Im Gegensatz zu ProGuard ist es ein kommerzielles Produkt.

R8 [18] ist ebenfalls ein Code-Minifizierer, um die Größe von Installationsdateien (APK-Dateien) zu reduzieren. Er wurde in Android Studio 3.3 eingeführt und kann ebenfalls zur Obfuskation verwendet werden. Im Vergleich zu ProGuard arbeitet es schneller und kann den Code stärker reduzieren.

Obfuscapk [1] ist ein modulares Werkzeug für Python zur Obfuskation von Android-Apps, ohne deren Quellcode zu benötigen, da apktool [51] verwendet wird, um die ursprüngliche apk-Datei zu dekompilieren und eine neue Anwendung zu erstellen, nachdem einige Obfuskationstechniken auf den dekompilierten Smali-Code, die Ressourcen und das Manifest angewendet wurden.

Dasho [25] ist ein weiteres kommerzielles Tool zum Schützen von Android-, Java- oder Webapplikationen.

3.3 iOS

Anwendungen können unter iOS mit Object-C und Swift geschrieben werden [14]. Der Code kann in nativen Code kompiliert werden. Seit XCode 7 kann der Code auch in BitCode umgewandelt werden, eine der drei Darstellformen der Intermediate Representation in LLVM. Zum Schutz können folgende Produkte verwendet werden:

Sirius [42] ist ein Open Source-Obfuskator, der in Swift geschrieben Projekte verschleiern kann.

Swift Shield [45] ist ein Werkzeug, das zufällige und unumkehrbare verschlüsselte Namen für die Typen und Methoden von iOS-Projekten einschließlich Bibliotheken von Drittanbietern erzeugt.

DexProtector [29] ist ein kommerzielles Produkt zum Schützen und zur Obfuskation von Android und iOS Anwendungen. Es unterstützt Anwendungsverschlüsselung, Schutz vor Analyse und Cracking und bietet Kopierschutz.

Promon SHIELD [43] ist ein kommerzielles Produkt, dass Apps vor Angreifern schützt. Es unterstützt auch Obfuskation für JavaScript, iOS und Android.

4 Vorteile und Nachteile von Obfuskation

Lutkevich [31], AtomicRogue1 [3] und Bridges [7] beschreiben einige Vor- und Nachteile dieser Technik in ihren Artikeln.

4.1 Vorteile

Die Hauptvorteile von Obfuskation sind die folgenden Argumente:

Geheimhaltung Die Verschleierung verbirgt die wertvollen Informationen, die im Code enthalten sind. Dies ist ein Vorteil für legitime Organisationen, die ihren Code vor Konkurrenten und Angreifern schützen wollen. Umgekehrt nutzen böswillige Akteure die Geheimhaltung der Verschleierung aus, um ihren bösartigen Code zu verstecken.

Effizienz Einige Verschleierungstechniken, wie z. B. das Entfernen von ungenutztem Code verkleinern das Programm und es wird weniger ressourcenintensiv. Ladezeiten können sich dadurch auch verkürzen.

Sicherheit Obfuskation ist eine eingebaute Sicherheitsmethode, die manchmal auch als Selbstschutz der Anwendung bezeichnet wird. Anstatt eine externe Sicherheitsmethode zu verwenden, funktioniert sie innerhalb des zu schützenden Objekts. Sie eignet sich gut für den Schutz von Anwendungen, die in einer nicht vertrauenswürdigen Umgebung ausgeführt werden und sensible Informationen enthalten. Einige Experten argumentieren, dass obfuskierter Code nicht unfehlbar ist. Wenn der Code begehrenswert genug ist, könnte er mit erheblichem Zeit- und Arbeitsaufwand entschlüsselt werden.

Verfolgung von illegalen Kopien Wenn verschleierter Code für ein Projekt verwendet wird, können alle Änderungen daran sofort identifizieren werden. Wenn dieser an mehrere Benutzer weitergegeben wurde, kann der Ursprung von illegalen Kopien sogar feststellen werden.

4.2 Nachteile

Obfuskation hat auch Nachteile. Zu diesen zählen die folgenden Punkte:

Einsatz in Malware wobei einige Technik besonders oft verwendet werden: Verschleierung von Code durch die Anwendung von XOR-Werten, Einfügen von toten Code, selbst modifizierende Programme sowie Neuanordnung der Befehle in Programmen.

Wartbarkeit Bei der Verschleierung wird ein bestimmter Code für andere im Wesentlichen unlesbar gemacht. Dies bietet zwar Sicherheit für den Code, erfordert aber auch ein gewisses Maß an Fachwissen bei der Wartung des Codes. Nur der Autor des Codes, ist in der Lage ihn anzupassen, nachdem er verschleiert worden ist.

Bugs Obfuskierter Code erschwert die Fehlersuche. Es wird auch für den Benutzer schwieriger, genaue Fehlerberichte einzureichen. Der Code kann auch

so unleserlich sein, dass es sogar für den ursprünglichen Programmierer schwierig sein kann, das wirkliche Problem zu finden.

Abbildung 2 zeigt ein Beispiel für einen Cross-Site-Scripting-Angriff, bei der der Code verschleiert wurde (Grafik entnommen aus Nunan et al. [36]).

```
1  http://www.trustedsite.com/search.html?type=<"<<sCrIpT>alert
2  (document.cookie)</sCrIpT><"<<sCrIpT>alert(document.cookie)
3  </sCrIpT>
4  http://www.trustedsite.com/search.html?type=%3C%22%3C%3C
5  %73%43%72%49%70%54%3E%61%6C%65%72%74%28%64
6  %6F%63%75%6D%65%6E%74%2E%63%6F%6F%6B%69%
7  65%29%3C%2F%73%43%72%49%70%54%3E%3C%22%3C
8  %3C%73%43%72%49%70%54%3E%61%6C%65%72%74%28
9  %64%6F %63%75%6D%65%6E%74%2E%63%6F%6F%6B%
10 69%65%29%3C%2F%73%43%72 %49%70%54%3E
```

Abbildung 2: Verschleierter Cross-Site-Scripting-Angriff

5 Obfuskation in Free/Libre Open Source Software

Wie der Name bereits vermuten lässt, bedeutet Open Source-Software, dass der Quellcode frei zugänglich sein muss. Nun stellt sich die Frage, ob verschleierter Quellcode als freier Quellcode gilt. Diese Frage wurde bereits in [9] gestellt. Laut der Free Software Foundation [13] ist eine Software eine freie Software wenn folgende 4 Freiheiten gegeben sind:

- Das Programm kann für jeden Zweck so ausgeführt werden, wie es gewünscht wird.
- Das Programm kann analysiert und so verändern werden, dass Berechnung nach Wunsch ausgeführt werden. Der Zugriff auf den Quellcode ist dafür eine Voraussetzung.
- Kopien dürfen verbreitet werden, um andere damit zu helfen. Kopien einer modifizierten Version dürfen an andere verteilt werden.
- Damit wird der ganzen Gemeinschaft die Chance gegeben, von den Änderungen zu profitieren. Der Zugriff auf den Quellcode ist dafür eine Voraussetzung.

In einem folgenden Absatz werden zusätzliche Information gelistet:

> **Die Freiheit, den Quellcode zu studieren und Änderungen vorzunehmen.** *Damit die Freiheiten 1 und 3 (die Freiheit, Änderungen vorzunehmen und die Freiheit, die geänderten Versionen zu veröffentlichen) sinnvoll sind, muss man Zugang zum Quellcode des Programms haben. Daher ist die Zugänglichkeit des Quellcodes eine notwendige Bedingung für freie Software. Verschleierter Quellcode ist kein echter Quellcode und zählt nicht als Quellcode.*

Die Open Source Initiative erwähnt dasselbe Thema in der Open Source Definition [39]:

> **Source Code.** *Der Quellcode muss in der bevorzugten Form vorliegen, in der ein Programmierer das Programm ändern würde. Absichtlich verschleierter Quellcode ist nicht erlaubt.*

Freizügigere Freie Software/Open Source-Lizenzen, erlauben es die Software proprietär werden zu lassen. In diesem Fall darf die Software ohne Zugriff auf den Quellcode weitergeben werden. Dies schließt Obfuskation mit ein. Im Zweifelsfall sollte die genau Definition der gewählten Lizenz genauer betrachtet werden.

6 Dekompilierung

Ein Dekompilierer ist ein Computerprogramm, das eine ausführbare Datei als Eingabe nimmt und versucht, eine Quelldatei zu erstellen, die erfolgreich rekompiliert werden kann. Während Dekompilierer normalerweise verwendet werden, um Quellcode aus binären ausführbaren Dateien wieder zu erzeugen, gibt es auch Dekompilierer, die bestimmte binäre Dateien in für den Menschen lesbare und editierbare Dateien verwandeln. Wie bei Rolles [46] beschrieben, ergeben sich speziell bei Maschinencode einige Herausforderungen, die durch dem Verlust von semantischen Informationen während des Kompilierungsprozesses entstehen. Die gleichen Probleme betreffen auch Quellcodes die verschleiert wurden. Verwaltete Sprachen wie z.B. Java oder C# bewahren oft die Namen von Variable und daher ist es einfach, lesbare Namen anzuzeigen. Dadurch kann der dekompilierte Maschinencode schneller verstanden werden.

Die meisten Kompilierer, die Maschinencode erstellen, verlieren diese Informationen beim Kompilieren, außer wenn Debug-Informationen erhalten bleiben. Jaffe et al. [26] betrachten daher die Dekompilierer-Ausgabe als eine verrauschte Verzerrung des ursprünglichen Quellcodes. Der ursprüngliche Quellcode

wurde in die Dekompilierer-Ausgabe transformiert. Unter Verwendung dieses verrauschten Kanalmodells wenden sie Standardverfahren der statistischen maschinellen Übersetzung an ,um natürliche Bezeichner auszuwählen.

Desnos and Gueguen [10] beschreiben die Dekompilierung unter Android. Sie erwähnen, dass es leichter ist als bei Maschinencode, da viele semantische Informationen im Bytecode bestehen bleiben. Es können nützliche Details über Variablen, Felder, Methoden erhalten werden. Für Methoden können beispielsweise Signaturen erstellt werden. Es können auch Android-Berechtigung verwendet werden, um herauszufinden, wo eine bestimmte Methode in einer Anwendung verwendet wird. Der Analyseteil erlaubt es den Kontrollflussgraphen zu extrahieren. Dieser besteht aus Basisblöcken und kann aufgrund der virtuellen Maschine nicht dynamisch verändert werden. Er wird verwendet, um die verschiedenen möglichen Pfade einer Anwendung darzustellen.

Die OWASP Foundation [40] beschreibt Manipulation und Reverse Engineering unter iOS. Es wird erwähnt, dass Reverse Engineering möglich ist, aber Einschränkungen beachtet werden müssen. Einerseits lassen sich Apps, die in Objective-C und Swift programmiert sind, gut disassemblieren. In Objective-C werden Objektmethoden über dynamische Funktionszeiger namens "Selektoren" aufgerufen, die zur Laufzeit namentlich aufgelöst werden. Der Vorteil der Namensauflösung zur Laufzeit ist, dass diese Namen in der endgültigen Binärdatei intakt bleiben müssen, wodurch die Disassemblierung besser lesbar wird. Dies bedeutet auch, dass keine direkten Querverweise zwischen Methoden im Disassembler verfügbar sind und die Konstruktion eines Flussgraphen eine Herausforderung darstellt. Im Handbuch werden statische und dynamische Analysemethoden erklärt.

Lu and Debray [30] zeigen, dass auch bei Web-Anwendungen der verschleierte JavaScript-Code vereinfacht werden kann. Sie präsentieren einen semantikbasierten Ansatz zur automatischen Deobfuskation von JavaScript-Code. Sie verwenden dynamische Analyse- und Programm-Slicing-Techniken, um die Verschleierung zu vereinfachen und die zugrunde liegende Logik des JavaScript-Codes in Webseiten aufzudecken. Der Ansatz macht keine Annahmen über die Struktur der Obfuskation. Experimente mit einer Prototyp-Implementierung zeigen, dass diese Technik auch gegen stark verschleierte JavaScript-Programme wirksam ist.

7 Experiment: Verschleierte Webanwendung

Die gewählte Anwendung ist in HTML/CSS + Vanilla JavaScript geschrieben d.h. es wurden keine externen Ressourcen oder Bibliotheken verwendet. Das

Projekt ist ein Open Source-Projekt [52], bei dem ein Taschenrechner implementiert wurde (siehe Abbildung 3). Der JS-Code ist 115 Zeilen lang und in einer einzelnen Datei *index.js* untergebracht. Es wurde dieses Projekt aufgrund der Einfachheit und Kürze gewählt und der Tatsache, dass es keine externen Dateien benötigt und direkt im Webbrowser ausgeführt werden kann. Es wurde kein eigenes Projekt implementiert, da hier nur die Obfuskation im Vordergrund steht.

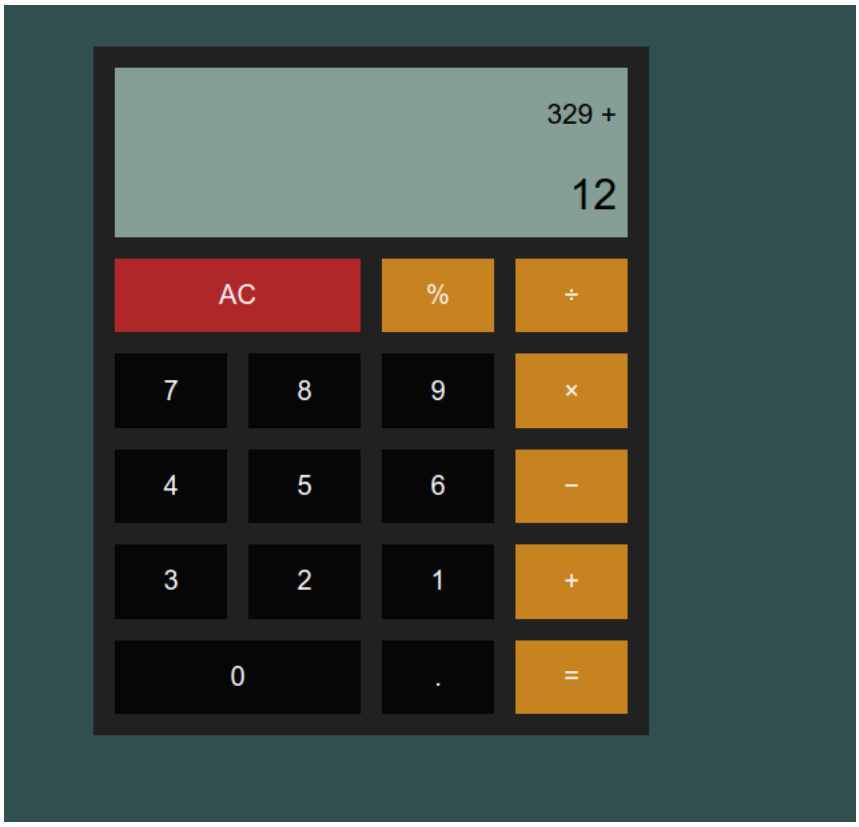

Abbildung 3: Webapplikation: Taschenrechner

7.1 Verwendete Bibliotheken

Es wurden die nicht verschleierte *index.js* mit 5 anderen Varianten verglichen, die mit verschiedenen Open Source-Bibliotheken mit Standardkonfigurationen verschleiert und minimiert wurden:

UglifyJS3 [6] ist ein JavaScript-Parser, Minimierer, Kompressor und Verschönerungs-Toolkit.

Terser [50] ist ein Parser-, Mangler- und Kompressor-Toolkit für JavaScript ES6+.

JavaScript obfuscator [37] ist ein leistungsfähiger kostenloser Obfuskator für JavaScript.

Babel Minify [4] ist ein ES6+-fähiger Minimierer, der auf der Toolchain von Babel basiert.

Non-Alphanumeric JS obfuscator [23] wandelt Javascript-Code in nahezu unumkehrbaren, nicht-alphanumerischen, obfuskierten Code um.

7.2 Gemessene Faktoren

Es wurde die Komplexität von *index.js* mit der Open Source-Bibliothek escomplex [12] gemessen. Komplexität beschreibt eine Gruppe von Merkmalen von Code. Diese Merkmale konzentrieren sich alle darauf, wie der Code mit anderen Teilen des Codes interagiert. Der Code wird an escomplex in Form von Syntaxbäumen übergeben, die mit esprima [27], einem populären JavaScript-Parser, generiert wurden. Es wurde auch der relative Größenunterschied der Dateien in Prozent aufgezeichnet. Die folgende Liste beschreibt alle gemessenen Metriken:

Mittlere logische LOC (LLOC)
LLOC ist die Anzahl an logischen Programmzeilen pro Funktion (Zählung der imperativen Anweisungen).

Mittlere Anzahl an Parameter pro Funktion (PAR)
PAR wird statisch aus der Funktionssignatur ausgewertet. Niedriger ist besser.

Mittlere zyklomatische Komplexität [33] pro Funktion (ZYK)
ZYK ist eine Zählung der Anzahl von Zyklen im Programmablaufsteuerungsgraphen. Niedriger ist besser.

Mittlere Halstead-Aufwand [21] pro Funktion(HSA)
HSA ist ein Maß zur Bestimmung des mentalen Aufwands, der zur Entwicklung oder Aufrechterhaltung eines Programms erforderlich ist. Niedriger ist besser.

Wartbarkeitsindex [38] (WI) ist eine logarithmische Skala von negativ unendlich bis 171, berechnet aus den logischen Codezeilen, der zyklomatischen Komplexität und dem Halstead-Aufwand. Höher ist besser.

Dateiengrößenunterschied (DGU) in Abhängigkeit zu der originalen *index.js*

7.3 Ergebnisse

Die Tabelle 1 stellt die Ergebnisse des Vergleichs der Orignaldatei mit 5 verschiedenen Bibliotheken dar. Non-Alphanumeric ist hier nur als Proof of Concept zu sehen, da es in der Produktion bei Unternehmen nicht verwendet wird. Es soll viel mehr gezeigt werden, dass eine Obfuskation möglich ist, bei der keine alphanumerischen Zeichen verwendet werden.

	Metriken					
Bibliothek	LLOC	PAR	ZYK	HSA	WI	DGU
Ohne	8.8	0.4	2.8	1386.7	110.8	0 %
UglifyJS3	5.2	0.4	2.4	1402.3	119.3	-38 %
Terser	11.5	0	4	2997.5	103.7	-61 %
JS Obfuscator	6.6	0.9	2.4	2751.8	113.2	21 %
Babel Minify	5.2	0.4	2.6	1435.2	119.2	-39 %
Non-Alphanumeric	/	/	/	/	/	492 %

Tabelle 1: Ergebnisse der Messung

Es fallen sofort die fehlenden Werte für Non-Alphanumeric auf. Es gibt keine Werte, da die Bibliothek zur Messung eine Fehlermeldung bei diesen Datensatz anzeigt. Die rekursive Implementierung der Bibliothek konnte mit der tiefen Verschachtelung im Code nicht umgehen. Dies deutet trotz fehlender Metriken auf eine hohe Komplexität hin. Der hohe Anstieg der Dateigröße kann mit der Lookup-Tabelle erklärt werden, die bei dieser Technik verwendet wird und immer im Programmcode eingebunden werden muss. Dadurch hat der kompilierte Code immer eine bestimmte Mindestgröße.

Die durchschnittliche logische LOC ist niedriger bei den Bibliotheken außer bei Terser. Diese Bibliothek kommt auch komplett ohne Funktionsparameter aus, wie man an der zweiten Metrik erkennen kann. JS Obfuscator hat in dieser Metrik einen mehr als doppelt so hohen Wert.

Die durchschnittliche zyklomatische Komplexität ist bei allen Varianten ähnlich außer bei Terser. Es fällt hier bereits deutlich auf, dass die Terser-Implementierung sich deutlich von den anderen Bibliotheken unterscheidet. Dies wird nochmal anhand des durchschn. Halstead-Aufwand bestätigt, der bei Terser und bei JS Obfuscator höher ist als bei den anderen Varianten.

Der Wartbarkeitsindex hat sich in allen Fällen verbessert außer bei Terser,

wo er leicht gesunken ist. Durch die Minimierung des Codes ist diese Metrik nicht sehr deutlich sichtbar, da alle Leerzeichen in den Dateien entfernt wurden. Diese sind allerdings subjektiv gesehen für die Wartbarkeit sehr wichtig, daher ist diese Metrik hier nicht sehr aussagekräftig.

Die drei Bibliotheken Uglify, Terser und Babel Minify unterstützen alle das Minimierung von Quellcode und erreichen daher in der letzten Metrik gute Werte. Negative Werte bedeuten eingesparten Speicherplatz. JS Obfuscator ist ein reiner Obfuskator, wie man deutlich an der Dateigröße erkennen kann: die Größe ist um 21 % gestiegen. Bei der Minimierung hat Terser mit einer Einsparung von 61 % am besten abgeschnitten. Das Ergebnis kann in Listing 1 betrachtet werden.

Listing 1: Ausgabe des Terser Manglers

```
const e=document.querySelector("#calculator"),a=
    document.querySelector("#formula-screen"),c=
    document.querySelector("#output-screen");document.
    querySelector("#equals"),document.querySelector("#
    ac"),document.querySelector("#equals");let t,o="",s
    ="";function r(){let e;const a=parseFloat(o),c=
    parseFloat(s);switch(t){case"%":e=a%c;break;case"/"
    :e=a/c;break;case"*":e=a*c;break;case"-":e=a-c;
    break;case"+":e=a+c;break;default:return}s=e,t=void
     0,o=""}function l(){c.value=s,a.value=t?`${o}${t}`:""}e
    .addEventListener("click",(e=>{if(e.target.matches(
    "button.button")){let d=e.target.value;switch(d){
    case"AC":console.log("all clear"),a.value="",c.
    value="0",t=void 0,o="",s="";break;case"%":case"/":
    case"*":case"-":case"+":console.log("append a math
    operator"),u=d,""!==s&&(""!==o&&r(),t=u,o=s,s=""),l
    ();break;case"0":case"1":case"2":case"3":case"4":
    case"5":case"6":case"7":case"8":case"9":case".":
    console.log("append a number"),"."===(n=d)&&s.
    includes(".")||(s=s.toString()+n.toString()),l();
    break;case"=":console.log("evaluate the expression"
    ),r(),l();break;default:console.log("something went
     wrong")}}var n,u}));
```

7.4 Fazit

Aus menschlicher Sicht ist Non-Alphanumeric am schwersten zu lesen, gefolgt von JS Obfuscator. Das Experiment hat gezeigt, dass eine Obfuskation nicht

immer auch zu einer Codereduzierung führen muss. Alle Bibliotheken wurden mit Standardparametern ausgeführt. Die gemessenen Metriken ändern sich unter Umständen stark, wenn andere Parameter gewählt werden. Die Verwendung von moderneren JavaScript-Features hat auch eine Auswirkung auf das Ergebnis. Einige der hier getesteten Bibliotheken unterstützen ES6-Features und sind für diese Features optimiert, die hier allerdings nicht getestet wurden. Eine Verbesserung wäre die Testwiederholung mit verschiedenen Quellcodes.

7.5 Obfuskation auf der Serverseite

Obfuskation wird auf der Serverseite nicht so häufig eingesetzt wie auf der Clientseite, da der Code auf dem Server nicht vom Benutzer im Quellcode analysiert werden kann. Auf der Clientseite erfolgt die Obfuskation meist als Teil des Kompilierungsvorgangs in einem Buildwerkzeug wie zum Beispiel Parcel oder Rollup.js. Auf der Serverseite muss die Ausgabe zuerst in einem Buffer gespeichert werden, verschleiert werden und dann auf der Webseite ausgegeben werden. Der User Macmillan [32] der Seite Stackoverflow bringt hierzu das Code-Beispiel 4 wie dies beispielsweise in PHP durch Steuerung der Ausgangspufferung gelöst werden kann.

```
<?php
// Start output buffering.
ob_start();

?>
function redirect() {
var r = <?php echo $rvar; ?>;
  if (r) {
    window.location = prepare("<?php echo $redirect; ?>");
  }
}
<?php

// Get all the output captured in the buffer, and turn off output buffering.
// This will be a string.
$js_source = ob_get_flush();

// Obfuscate the $js_source string, by passing it to the obfuscator
// library you found.
$obfuscated_js_source = obfuscator5000($js_source);

// Send the source code to the browser by echoing it out.
echo $obfuscated_js_source;
```

Abbildung 4: Serverseitige Obfuskation mit PHP

Im Bezug auf mobile Android-Anwendungen kann die serverseitige Obfuskation für einen anderen Zweck verwendet werden. Piao et al. [41] schlagen eine Verschleierungstechnik vor, die auf einem Client/Server-Modell mit einmaliger Übermittlung des geheimen Schlüssels über einen Kurznachrichtendienst oder

ein Netzwerkprotokoll basiert. Das Hauptkonzept besteht darin, die wichtigste Ausführungsdatei durch Obfuskation auf dem Server zu speichern, sodass ein Programm, das Kernroutinen ausführen muss, diese Routinen vom Server anfordern muss. Auf diese Weise können Android-Apps vor Reverse Engineering geschützt werden.

8 Zusammenfassung

In dieser Arbeit wurde zuerst der Begriff Obfuskation definiert und von den ähnlichen Begriffen Encodierung, Verschlüsselung und Hashing abgegrenzt. Danach wurden verschiedene Techniken der Code-Obfuskation aufgelistet, wobei die Transformationen in rechnerischen Umformen sowie Transformationen der Datenstruktur unterschieden wurde. Dann wurden die Verschleierung bei Wep-Apps betrachtet, wobei speziell die Technologien HTML und JavaScript betrachtet wurden. Es wurden auch Softwarebibliotheken für die Plattformen Android und iOS vorgestellt. Nachfolgend wurden einige Vor- und Nachteile von Obfuskation geschildert und den Einsatz von Verschleierung in freier Software betrachtet. In einem Experiment, bei dem eine Webanwendung verschleiert wurde, wurden verschiedene Komplexitätsmetriken gemessen. Es stellte sich heraus, dass man Bibliotheken zur Minimierung und Obfuskation unterscheiden kann und die Verschleierungstechniken von Bibliothek zu Bibliothek sehr unterschiedlich implementiert sind. Obfuskation führt auch nicht immer automatisch zur Verkleinerung der Dateigröße. Zuletzt wurde noch beschrieben, wie auf der Serverseite eine Obfuskation implementiert werden kann.

9 Zukünftige Entwicklungen

In den letzten Jahren wurde untersucht, ob Obfuskation als Baustein für Kryptografie verwendet werden kann. Einer dieser Forschungszweige nennt sich Indistinguishability obfuscation [15] und ist ein kryptographisches Baustein, das einen formalen Begriff der Programmverschleierung bietet. Informell gesehen verbirgt die Verschleierung die Implementierung eines Programms, während der Benutzer es trotzdem ausführen kann. Es konnte bis jetzt noch nicht konstruiert werden.

Die Black-Box-Verschleierung war ein weiteres vorgeschlagenes kryptographischer Baustein, die es erlauben sollte, ein Computerprogramm so zu verschleiern, dass es unmöglich wäre, irgendetwas darüber zu bestimmen, außer seinem Eingabe- und Ausgabeverhalten. Black-Box-Verschleierung hat sich als unmöglich erwiesen [5].

Bei Deobfuskatoren werden in der Zukunft maschinelles Lernen verwendet, um den Quellcode noch lesbarer zu machen. Eine aktuelle Forschungsarbeit zu diesem Thema ist z.B. Neutron von Liang et al. [28], in der ein neuronalen Dekompilierer implementiert wurde.

Listings

Literatur

[1] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2020.100403. URL `http://www.sciencedirect.com/science/article/pii/S2352711019302791`.

[2] Apple. Security | apple developer documentation. `https://developer.apple.com/documentation/security`, 2021. [Online; abgerufen am 6.05.2021].

[3] Benutzer AtomicRogue1. What is obfuscation? - geeksforgeeks. `https://www.geeksforgeeks.org/what-is-obfuscation`, 2020. [Online; abgerufen am 6.05.2021].

[4] babel. babel/minify: An es6+ aware minifier based on the babel toolchain (beta). `https://github.com/babel/minify`, 2021. [Online; abgerufen am 6.05.2021].

[5] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Annual international cryptology conference*, pages 1–18. Springer, 2001.

[6] Mihai Bazon. Uglifyjs 3. `https://github.com/mishoo/UglifyJS`, 2021. [Online; abgerufen am 6.05.2021].

[7] The Digital Bridges. What is obfuscation? - geeksforgeeks. `https://www.thedigitalbridges.com/obfuscated-code-advantages-security`, 2017. [Online; abgerufen am 6.05.2021].

[8] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[9] ctrl-alt delor. licensing - code obfuscation in foss source code - open source stack exchange. `https://opensource.stackexchange.com/a/8458`, 2019. [Online; abgerufen am 6.05.2021].

[10] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, pages 77–101, 2011.

[11] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 4(8), 2010.

[12] escomplex. escomplex/escomplex: Software complexity analysis of javascript-family abstract syntax trees. `https://github.com/escomplex/escomplex`, 2017. [Online; abgerufen am 6.05.2021].

[13] Free Software Foundation. What is free software? - gnu project - free software foundation. `https://www.gnu.org/philosophy/free-sw.en.html`, 2021. [Online; abgerufen am 6.05.2021].

[14] Cristian González García, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. Swift vs. objective-c: A new programming language. *IJIMAI*, 3(3):74–81, 2015.

[15] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3): 882–929, 2016.

[16] Google. Implement security by design for your apps | android developers. `https://developer.android.com/security`, 2021. [Online; abgerufen am 6.05.2021].

[17] Google. Android runtime (art) und dalvik | android open source project. `https://source.android.com/devices/tech/dalvik`, 2021. [Online; abgerufen am 6.05.2021].

[18] Google. Shrink, obfuscate, and optimize your app | android developers. `https://developer.android.com/studio/build/shrink-code`, 2021. [Online; abgerufen am 6.05.2021].

[19] Guardsquare. Dexguard | guardsquare. `https://www.guardsquare.com/dexguard`, 2021. [Online; abgerufen am 6.05.2021].

[20] Guardsquare. Proguard | guardsquare. `https://www.guardsquare.com/proguard`, 2021. [Online; abgerufen am 6.05.2021].

[21] Maurice H Halstead. Toward a theoretical basis for estimating programming effort. In *Proceedings of the 1975 annual conference*, pages 222–224, 1975.

[22] Mario Heiderich, Eduardo Alberto Vela Nava, Gareth Heyes, and David Lindsay. *Web Application Obfuscation:'-/WAFs.. Evasion.. Filters//alert (/Obfuscation/)-'*. Elsevier, 2010.

[23] https://mrpapercut.com/. Non-alphanumeric javascript obfuscator. `https://mrpapercut.com/sites/obfuscator`, 2016. [Online; abgerufen am 6.05.2021].

[24] Jung-Chang Huang and Tau Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122)*, pages 244–248. IEEE, 1999.

[25] Idera Inc. Dasho | preemptive. `https://www.preemptive.com/products/dasho`, 2021. [Online; abgerufen am 6.05.2021].

[26] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.

[27] jquery. jquery/esprima: Ecmascript parsing infrastructure for multipurpose analysis. `https://github.com/jquery/esprima`, 2021. [Online; abgerufen am 6.05.2021].

[28] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 4(1):1–13, 2021.

[29] Licel. Dexprotector - mobile application security | cryptobox rasp. `https://dexprotector.com`, 2021. [Online; abgerufen am 6.05.2021].

[30] Gen Lu and Saumya Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 31–40. IEEE, 2012.

[31] Ben Lutkevich. What is obfuscation and how does it work? `https://searchsecurity.techtarget.com/definition/obfuscation`, 2021. [Online; abgerufen am 6.05.2021].

[32] Dane Macmillan. php - how to obfuscate javascript on the server side? - stack overflow. `https://stackoverflow.com/a/25246021`, 2014. [Online; abgerufen am 6.05.2021].

[33] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

[34] Merriam-Webster. Obfuscate | definition of obfuscate by merriam-webster. https://www.merriam-webster.com/dictionary/obfuscate, 2021. [Online; abgerufen am 10.05.2021].

[35] Daniel Miessler. Hashing vs. encryption vs. encoding vs. obfuscation | daniel miessler. https://danielmiessler.com/study/encoding-encryption-hashing-obfuscation, 2020. [Online; abgerufen am 6.05.2021].

[36] Angelo Eduardo Nunan, Eduardo Souto, Eulanda M Dos Santos, and Eduardo Feitosa. Automatic classification of cross-site scripting in web pages using document-based and url-based features. In *2012 IEEE symposium on computers and communications (ISCC)*, pages 000702–000707. IEEE, 2012.

[37] obfuscator.io. javascript-obfuscator/javascript-obfuscator: A powerful obfuscator for javascript and node.js. https://github.com/javascript-obfuscator/javascript-obfuscator, 2021. [Online; abgerufen am 6.05.2021].

[38] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society, 1992.

[39] opensource.org. What is free software? - gnu project - free software foundation. https://opensource.org/osd, 2007. [Online; abgerufen am 6.05.2021].

[40] OWASP. ios tampering and reverse engineering. https://github.com/OWASP/owasp-mstg/blob/master/Document/0x06c-Reverse-Engineering-and-Tampering.md, 2021. [Online; abgerufen am 6.05.2021].

[41] Yuxue Piao, Jin-Hyuk Jung, and Jeong Hyun Yi. Server-based code obfuscation scheme for apk tamper detection. *Security and Communication Networks*, 9(6):457–467, 2016.

[42] Polidea. Polidea/siriusobfuscator. https://github.com/Polidea/SiriusObfuscator, 2018. [Online; abgerufen am 6.05.2021].

[43] Promon. Application protection - mobile in-app protection | promon. https://promon.co/products/mobile-app-protection, 2021. [Online; abgerufen am 6.05.2021].

[44] Purplesec. 2021 cyber security statistics: The ultimate list of stats, data & trends | purplesec. https://purplesec.us/resources/cyber-security-statistics, 2021. [Online; abgerufen am 6.05.2021].

[45] Bruno Rocha. rockbruno/swiftshield: Swift obfuscator that protects ios apps against reverse engineering attacks. https://github.com/rockbruno/swiftshield, 2021. [Online; abgerufen am 6.05.2021].

[46] Rolf Rolles. decompilation - why are machine code decompilers less capable than for example those for the clr and jvm? - reverse engineering stack exchange. https://reverseengineering.stackexchange.com/a/312, 2013. [Online; abgerufen am 7.05.2021].

[47] Spencer Rugaber, Kurt Stirewalt, and Linda M Wills. The interleaving problem in program understanding. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 166–175. IEEE, 1995.

[48] statista. Mobile operating systems' market share worldwide from january 2012 to october 2020. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009, 2021. [Online; abgerufen am 6.05.2021].

[49] sucuri.net. Owasp top 10 security vulnerabilities 2021 | sucuri. https://sucuri.net/guides/owasp-top-10-security-vulnerabilities-2021/, 2021. [Online; abgerufen am 6.05.2021].

[50] Terser. terser · javascript parser, mangler and compressor toolkit for es6+. https://terser.org, 2021. [Online; abgerufen am 6.05.2021].

[51] R Winsniewski. Android–apktool: A tool for reverse engineering android apk files. *Retrieved February*, 10:2020, 2012.

[52] Jun YuHuang. Junyuhuang/vanilla-js-todo-app. https://github.com/JunYuHuang/vanilla-js-todo-app, 2020. [Online; abgerufen am 6.05.2021].

[53] Bo Zhao, Yingying Li, Lin Han, Jie Zhao, Wei Gao, Rongcai Zhao, and Ramin Yahyapour. A practical and aggressive loop fission technique. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 66–75. Springer, 2018.