Information Security

Code Obfuscation im Test

Erzeugung von Testdaten mittels LLVM zum Analysieren, Trainieren und Auswerten von Software zur Erkennung von Obfuscation

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

eingereicht von

Michael Kraftl

1710619834

im Rahmen des

Studienganges Information Security an der Fachhochschule St. Pölten

Betreuung

Betreuer/in: FH-Prof. DI Dr Sebastian Schrittwieser, Bakk.

St. Pölten, 17. Mai 2019 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

*

Fachhochschule St. Pölten GmbH, Matthias Corvinus-Straße 15, A-3100 St. Pölten,T: +43 (2742) 313 228, F: +43 (2742) 313 228-339, E:office@fhstp.ac.at, I:www.fhstp.ac.at

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/-Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die mich während meines Studiums und bei meiner Abschlussarbeit unterstützt haben.

Zuerst gebührt der Dank meiner Familie, die mich über sechs Jahre bei meinem Vorhaben, den Studiengang IT-Security zu absolvieren, immer unterstützt hat. Ein ganz besonderer Dank geht an meine Frau Anita, die mich immer wieder motiviert hat, diese lange Zeit durchzuhalten. Auch meine Diplomarbeit wurde mit ihrer Gründlichkeit Korrektur gelesen.

Ebenfalls bedanken möchte ich mich bei meinem Betreuer FH-Prof. DI Dr Sebastian Schrittwieser, Bakk., der meine Arbeit betreut und begutachtet hat. Dabei ist er mir mit hilfreichen Tipps und Anregungen zur Seite gestanden.

Auch meinem Vorgesetzten, DI Christian Perschl, danke ich für die Rücksichtnahme und das Verständnis, das er aufgebracht hat, wenn ich mir für die FH Urlaub genommen habe und für ihn nicht zur Verfügung stand.

Zu guter Letzt geht auch ein Dank an meine Studienkollegen Stefan Hagl, Dominik Handl, Stefan Karner, Robert Prinz und Philip Scheidl. Es war mir immer eine Freude wie wir uns gegenseitig bei Gruppenarbeiten ergänzt haben.

Michael Kraftl

Kurzfassung

Wenn es nicht gerade um Open-Source-Software geht, wollen Autoren von Computerprogrammen oft ihren Code schützen. Da Software in den meisten Ländern nicht patentierbar ist, greifen die Programmierer zu anderen technischen Methoden. Code-Obfuscation ist eine gebräuchliche Methode, um Code mittelfristig vor ungewollter Analyse zu schützen. Diese Technik verwenden Programme, wie zum Beispiel der VoIP Client Skype, aber auch Malware versucht deren Funktion so zu verstecken. Um zu erkennen ob eine Software verschleiert ist, werden aktuell verschiedene Techniken, wie statistische Methoden, aber auch maschinelles Lernen getestet. Dafür werden Testdaten in größerem Umfang benötigt. Diese Arbeit sucht Wege, um diese Testdaten einfach und effizient zu erzeugen. Dazu werden vorhandene Tools für Verschleierung evaluiert und getestet. Dabei werden die, auf der Compiler-Suite LLVM basierenden Verschleierung-Suites „Obfuscator-LLVM“ und „Hikari“ genauer betrachtet. Als Quelle für die Eingangsdaten werden die Quellpakete einer Linux Distribution verwendet. Nach der Übersetzung dieser Pakete ist es erforderlich, eine Nachverarbeitung der entstandenen binären Dateien vorzunehmen. Als Ergebnis dieser Arbeit stehen rund 700 aus Quellpaketen kompilierte Programme in verschiedenen Verschleierungstechniken zur Verfügung. Diese werden für die Entwicklung von Software für die Erkennung von Code Obfuscation eingesetzt.

Abstract

When it comes to open source software, computer program authors often want to protect their code. Since software is not patentable in most countries, the programmers use other technical methods. Code obfuscation is a common way to protect code from unwanted analysis in the medium term. This technique is used by programs, such as the VoIP client Skype, but also malware tries to hide their functionality by this technique. In order to detect whether a software is obfuscated or not, currently various techniques, such as statistical methods or machine learning are under evaluation. Therefore, test data is needed on a larger scale. This work tries to find ways to generate test data easy and efficient. Therefore, existing tools for obfuscation are evaluated and tested. It looks more closely to "obfuscator-LLVM" and "Hikari" obfuscator suites based on the LLVM compiler suite. The sources of a Linux distribution are taken as input data for the obfuscator suites. After the translation of these packages, it was necessary to post process the resulting binary files. As a result of this work, around 700 programs in various obfuscation techniques are available. These are used for the development of software to detect code obfuscation.

Inhaltsverzeichnis

1. Einleitung

Schon im Mittelalter beschäftigen sich die Menschen mit dem Schutz ihrer Erfindungen. Zu dieser Zeit gibt es den Begriff "Patent“ noch nicht und es werden sogenannte "Privilegien“ an Gewerbebetriebe vergeben. Der Inhaber dieser Privilegien ist alleinig für die Verwertung der Idee berechtigt. Das älteste Patentgesetz stammt aus Venedig aus dem Jahre 1474 [1] und enthält alle wesentlichen Kriterien heutiger Patentgesetze. Damalige gemeldete Erfindungen sind 10 Jahre gegen Nachahmung geschützt und dürfen nur von ihrem Anmelder verwertet werden. Erst 1877 [1] wird in Deutschland das Patentgesetz erlassen und 1898 folgt Österreich mit einem eigenen Gesetz. In Österreich stammt das aktuelle Patentgesetz aus dem Jahr 1970 [2]. Dies ist aber zu früh, um den Durchbruch der Personal Computer zu berücksichtigen. Im Jahr 1984 gibt es eine Patentgesetz-Novelle, die explizit ausschließt, dass „Programme zur Datenverarbeitung“ patentierbar sind. Auch mathematische Methoden sind ausgeschlossen. Aktuell können in Österreich Softwarepatente erteilt werden, wenn diese einen „technischen Charakter“ aufweisen. Nach der „Richtlinie zur Bearbeitung von Anmeldungen zu Computerimplementierten Erfindungen“ [3] ist, neben weiteren Bedingungen, der technische Charakter unbedingt erforderlich für die Schutzfähigkeit der Software. Neben dem Patent ist der Sourcecode aber immer, und das passiert automatisch, durch das Urheberrecht geschützt. Dieses Recht ist weltweit anerkannt und durchsetzbar. Das Urheberrecht verbietet das Kopieren des Sourcecodes, das heißt aber auch, dass bei geringer Modifikation des Sourcecodes keine Kopie mehr vorliegt. Aus diesem Grund ist der Schutz durch das Urheberrecht gering. Auf europäischer Ebene ist es seit 1973 durch das europäische Patentübereinkommen (EPÜ) nicht zulässig, für Software Patente zu erteilen und es gelten ähnliche Regeln wie in Österreich. In den Vereinigten Staaten von Amerika [4] ist es aktuell möglich, Software zu patentieren. Dies hat der Oberste Gerichtshof 1980 beschlossen. Aus den oben angeführten Fakten ist zu erkennen, dass der Schutz von Software als solcher weltweit nicht einheitlich geregelt ist. Da Hersteller von Software dennoch ihre verwirklichten Ideen und Erfindungen in diesem Bereich schützen möchten, versuchen sie andere Wege zu gehen. Heute sind mehrere Methoden bekannt, um Software vor ungewollter Verbreitung und Analyse zu schützen.

1.1. Schutz vor illegaler Softwareverbreitung

Gerade im Bereich der Computerspiele sind die Kosten für eine Produktion enorm. Um Entwicklungskosten von 100-200 Millionen US$ [5] wieder einzunehmen, ist es erforderlich, die illegale Verbreitung von Kopien der Software zu unterbinden. Dabei greifen die Hersteller oft zu Kopierschutz. Über die Jahre hinweg entstand allerdings ein „Katz und Maus“ Spiel zwischen Entwicklern von Kopierschutzsoftware und der Cracker-Szene. Als Cracker werden Personen bezeichnet, die versuchen den Kopierschutz von geschützten Programmen durch Analyse und Modifikation des Codes zu umgehen. Das Entfernen des Kopierschutzes funktioniert, mit dem nötigen Wissen, so einfach und zuverlässig, dass oft nur wenige Tage nach Erscheinen einer neuen Software eine modifizierte, frei kopierbare Variante auf einschlägigen Seiten verfügbar ist. Weitere Möglichkeiten die illegale Verbreitung von Programmen zu verfolgen sind Watermarking [6], Fingerprinting und Bithmarking [7]. Dabei geht es lediglich um die Wiedererkennung und Verfolgung der Software und nicht um die Verhinderung der Verbreitung. Diese Methoden können jedoch genutzt werden, um zum Beispiel das Hochladen von urheberrechtlich geschützten Werken auf Videoplattformen zu verhindern oder den Ursprung der illegalen Verbreitung festzustellen.

1.2. Schutz vor Analyse der Software

Die Gründe warum Software vor Analyse geschützt werden muss, sind unterschiedlich. Wie bereits in der Einleitung 1 erwähnt, kann der Schutz eines Algorithmus der Beweggrund sein. Ein weiterer Grund ist der in Kapitel 1.1 genannte Kopierschutz. Um diesen zu umgehen, ist eine Analyse des Binärcodes bzw. des dekompilierten Codes notwendig, um jene Algorithmen im Code zu finden, die das ungewollte Kopieren und Verbreiten der Software verhindern. Je offensichtlicher der Kopierschutz im Programm eingebracht ist, umso leichter ist das Entfernen. Um den Crackern die Arbeit zu erschweren, wird der Kopierschutz an mehreren Stellen eingefügt und die Funktionen werden verschleiert. Umso mehr der Code verschleiert wird, desto länger dauert die Analyse und die Umgehung des Schutzes. Damit ergibt sich eine Kosten-Nutzenrechnung für die Cracker und die Umgehung wird unter Umständen unattraktiv. Weitere Anwendung findet die Verschleierung, auch Obfuscation genannt, im Digital Rights Management wo kryptographische Schlüssel für die Offlineverfügbarkeit im Code vorhanden sind. Am Beispiel der DVD ist der Inhalt mittels Content Scramble System [8] verschlüsselt. Die Schlüssel dazu befinden sich in den Playern, welche entweder als Hardware oder als PC-Software ausgeführt sind. Bei beiden befinden sich die Schlüssel in der Software, die durch Analyse extrahiert werden können. 1999 ist es gelungen die Schlüssel auch zu extrahieren [9].
Eine weitere Anwendung ist das Verhindern von Betrug bei Computerspielen. Immer öfter ist reales

Geld mit im Spiel. Entweder finden Turniere statt, bei denen um Geld gespielt wird, oder der Hersteller bietet Zusatzleistungen gegen Geld an. Um die Business Model dahinter zu schützen, versuchen die Hersteller der Spiele die Manipulation zu verhindern oder zumindest zu erkennen. Die Algorithmen dahinter werden durch Verschleierung geschützt.
Wird Obfuscation auf Software angewendet ergeben sich nicht nur die vorher genannten Vorteile für die Hersteller. Im Allgemeinen muss durch Verschleierung damit gerechnet werden, dass durch die zunehmende Komplexität die Ausführungszeit und Größe der Programme steigen können. Dennoch setzen immer mehr Firmen, wie Microsoft, Apple, Intel und viele weitere [7] Obfuscation ein, um ihre Software vor illegaler Verwendung, Modifikation oder Analyse zu schützen.

1.3. Motivation

Wie in Absatz 1.1 und 1.2 beschrieben, kann Verschleierung benutzt werden, um das geistige Eigentum von Personen und Firmen zu schützen. Aber wie bei vielen Technologien kann auch diese für destruktive Vorhaben verwendet werden. Um den Analysten diverser Anti-Viren-Programm Hersteller die Arbeit zu erschweren, setzen Malware, Viren und Co schon lange Verfahren zur Verschleierung ein. Dabei werden unterschiedliche Ziele verfolgt. Zum Einen soll die Schadsoftware ungehindert von Virenscannern auf den Rechner des Opfers gelangen und ausgeführt werden, zum Anderen soll die Funktion vor den Analysten verborgen werden. Eine beliebte Methode ist das Einsetzen von Packern [10], die nachträglich das fertige Schadprogramm auf die unterschiedlichste Art und Weise verschleiern. Dadurch kann erreicht werden, dass ein und derselbe Virus bei Verwendung unterschiedlicher Packer oder Parameter desselben Packers unterschiedliche Signaturen hat. So kann sich ein bereits bekannter Virus durch eine andere Signatur erneut verbreiten, da das Anti-Viren-Programm oft signaturbasiert arbeitet. Um auf externe Packer zu verzichten, kann der Code auch entweder auf Quellcode, Zwischencode oder Assemblercode Ebene verschleiert werden. Die Vermeidung gleicher Signaturen erfolgt durch Diversität bei der Verschleierung.
Laut der Website AV-Test.org werden jeden Tag um die 350 k neue verdächtige und unerwünschte Programme gesichtet. Abbildung 1.1 zeigt den deutlichen jährlichen Anstieg der Schadprogramme. Bei dieser enormen Anzahl ist es schon bei unverschleierten Programmen ein beträchtlicher Aufwand, diese zu analysieren. Werden diese Schadprogramme verschleiert, steigt der Aufwand nochmals. Bei einer noch immer steigenden Anzahl von Samples pro Tag ist es für Virenautoren leicht, die Funktion ihrer Programme, im Hinblick von Kosten-Nutzen der Analyse, zu verschleiern. Und mit der zusätzlichen Funktion der Diversität können schon einmal analysierte Samples in ihrer modifizierten Form nicht mehr erkannt werden.

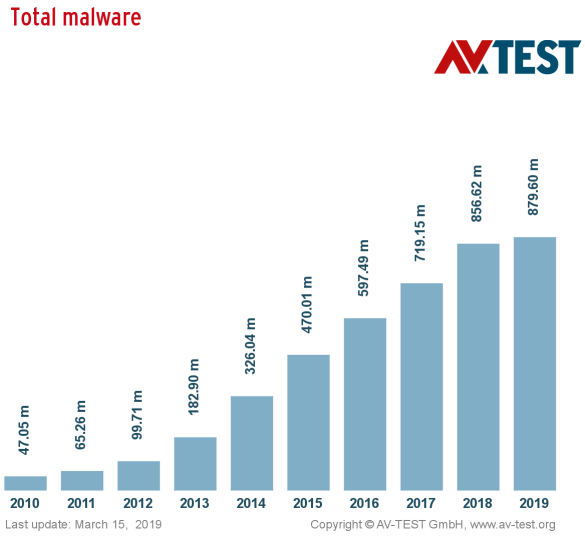

Abbildung 1.1.: Übersicht über die Anzahl der bekannten Malware Samples der letzten 10 Jahre nach AV-Test.org

Auch im Bereich der mobilen Software wie Android und iOS steigt die Zahl der verschleierten Programme an. Schon 2016 waren 50 % aller Android Play Store Pakete verschleiert [11]. Oft sind bei kostenlosen Programmen auch Mehrwertfunktionen für den Autor versteckt. Dabei werden persönliche Daten gesammelt oder ungewollte Funktionen des Mobiltelefons gesteuert. Diese sind bei verschleierten Programmen schwieriger zu entdecken.

Aktuell beschäftigt sich die Forschung mit verschiedenen Methoden, um zu evaluieren, ob ein Code verschleiert ist oder nicht. Um diese Methoden zu testen werden entsprechende Testdaten benötigt. Da normalerweise Programme nicht in beiden Varianten (verschleiert oder herkömmlich) vorliegen, ist es schwierig die entwickelten Methoden zu testen. Auch für das immer populärere maschinelle Lernen ist es notwendig, sowohl für die Lernphase als auch die Testphase, Daten in beiden Ausprägungen zur Verfügung zu haben.

Im Folgenden wird untersucht, auf welche Art und Weise Testdaten für die Entwicklung von Methoden zur Erkennung von verschleierter Software erzeugt werden können. Die Daten sollen mit einem existierenden Verschleierungsprogramm erzeugt werden, welches im Rahmen der Arbeit ausgewählt wird. Außerdem ist es notwendig, die geeignete Datenbasis für die Testdaten zu finden.

1.4. Gliederung

In Kapitel 2 werden die Grundlagen der Verschleierung von Programmcode, auf englisch Obfuscation, erklärt. Dabei wird auf den Unterschied zwischen den einzelnen Obfuscations-Techniken eingegangen. Für die Übersetzung der Linuxpakete wird LLVM verwendet. LLVM wird im Kapitel 4 beschrieben und erklärt, warum sich dieser Compiler besonders gut für die Verschleierung eignet. In den 2 anschließenden Abschnitten werden 2 Projekte (Obfuscator-LLVM 5 und Hikari 6) beschrieben und die Unterschiede aufgezeigt. Mit dem Hikari Projekt wurden die Linux Pakete für die Auswertung auch übersetzt. Kapitel 7 beschreibt die Herangehensweise und die Umsetzung der Arbeit. Mit Hilfe dieses Kapitels kann die Arbeit reproduziert werden. Im Kapitel 8 sind die Ergebnisse der Auswertungen zu finden. Das letzte Kapitel 9 ist eine Zusammenfassung.

2. Obfuscation

In der IT-Sicherheit ist „security by obsurity“ sehr umstritten. Es gibt jedoch Situationen in denen es nicht möglich ist, dies zu vermeiden. Im Speziellen geht es hier um den Schutz von Software vor ungewollter Analyse, Manipulation und im Weiteren auch um deren ungewollte Verbreitung. Bisher gibt es kein Verfahren, um Software ohne „security by obsurity“ zu schützen. Das liegt daran, dass jede Software von einem Rechner gelesen und ausgeführt werden muss. Dabei ergeben sich verschiedene Angriffsvektoren wie disassemblieren, debuggen oder Speicher auslesen, um nur einige zu nennen. Es gibt zwar einige Hardware basierte Schutztechniken, um diesen Vektoren entgegen zu wirken, aber diese erfordern spezielle Hardware und einen Prozess um Software an Hardware zu binden [7] . Daher ist im Bereich Softwareschutz die Methode „security by obsurity“ gebräuchlich. Da mit dieser Methode kein vollständiger Schutz erfolgen kann, wird hier mit einer Kosten-Nutzenrechnung gearbeitet. Ziel ist es, den Prozess des Reverse Reengineering soweit zu verlangsamen, dass es aus wirtschaftlicher Sicht uninteressant wird oder der Cracker aufgibt weil der Code zu komplex ist.

Um eine Software oder Teile davon ausreichend zu schützen, muss zuerst der Schutzbedarf erhoben werden. Dabei ist es wichtig, die Angriffsvektoren und -motivation zu kennen. Erst dann können die Methoden und der Aufwand für den Schutz festgelegt werden.

Firmen wie Microsoft, Apple, Skype und Intel [7] setzen Codeverschleierung ein, um ihre Produkte vor ungewollter Analyse und Manipulation zu schützen und damit auch ihr Knowhow. Verschleierung wird nicht nur von diesen Firmen angewendet, sondern auch aktiv weiterentwickelt. So hat Apple ein Patent auf eine Methode zur Verschleierung von Befehlssequenzen [12]. Andere arbeiten eng mit Firmen zusammen, welche sich auf Obfuscation spezialisiert haben. So hat PreEmptive Solutions Microsoft geholfen, Visual Studio zu schützen [7]. Die US Armee setzt bei allen Projekten beim Design und der Implementierung auf Anti Temper. So soll die US Waffentechnologie vor Reverse-Engineering aber auch Manipulation geschützt werden.

2.1. Grundlagen

Die Verschleierung von Code bedient sich Techniken aus dem Compilerbau. Wie auch die Optimierung von Code ist die Obfuscation mathematisch gesehen eine Transformation des Codes. Beide Varianten haben ein gemeinsames und ein oder mehrere abweichende Ziele. Das gemeinsame Ziel ist es, aus dem vorhandenen Programm $\mathcal{P}$ ein neues Programm $\mathcal{P}'$ zu erzeugen, das sich nach außen hin funktional gleich verhält. Optimierung das Ziel im Programm $\mathcal{P}'$ im Allgemeinen Laufzeit zu optimieren, zielt die Obfuscation darauf ab, es zu erschweren Informationen aus $\mathcal{P}'$ zu extrahieren. Das bezieht sich sowohl auf den Ablauf von als auch auf die Daten von $\mathcal{P}'$ [7, S. 201].
Die Transformation der Obfuscation kann in unterschiedlichen Phasen der Kompilierung erfolgen. Zum einen kann bereits der Quellcode derart modifiziert werden, dass zum Beispiel Funktionsnamen und Variablennamen verfremdet werden und damit keinen Rückschluss auf die Funktion des Codes möglich ist. Wird bei der Kompilierung ein Zwischencode erzeugt, kann zum Anderen an dieser Stelle eine Transformation erfolgen. Dies ist zum Beispiel bei dem Compiler LLVM möglich. Der Vorteil ist, dass hier nicht die Verschleierung für verschiedene Programmiersprachen separat entwickelt werden muss. Als dritte Variante steht die Obfuscation des Assembler- oder Maschinen-Codes zur Verfügung. Dabei werden vor allem hardwarenahe Verschleierungstechniken angewandt. Das Ziel der Obfuscation kann unterschiedlich sein: es kann die Absicht sein, einen Algorithmus zu schützen oder es kann notwendig sein, einen kryptographischen Schüssel zu schützen.

2.2. Vor- und Nachteile

In diesem Abschnitt werden die Vor- und Nachteile nach Babu et al. [13]von Verschleierung von Code aufgezeigt.

2.2.1. Vorteile

- **Schutz**: Obfuscation erschwert die statische und dynamische Codeanalyse.
- **Diversität**: mittels Obfuscation-Techniken ist es möglich, ein und dasselbe Programm in unterschiedlichen Ausprägungen zu übersetzen.
- **Günstig in der Anwendung**: Durch die Automatisierung des Transformationsprozesses ist die Anwendung leicht und günstig umzusetzen und in existierende Build-Systeme zu integrieren.
- **Plattformunabhängig**: Viele Transformationen können auf Quellcode oder Zwischencode angewendet werden und sind somit unabhängig von der Plattform.

2.2.2. Nachteile

- **Performance und Overhead**: Durch die angewendeten Techniken werden oft Daten oder Code zur Binärdatei hinzugefügt. Dadurch ergeben sich im Allgemeinen Nachteile in der Laufzeit. Die Dateigröße liegt, je nach angewendeter Technik, auch meist über der originalen Datei.
- **Kein Langzeitschutz**: Wie der Name schon sagt, bietet Verschleierung keinen dauerhaften Schutz gegen Reverse-Engineering. Mit genügend Aufwand können die Funktion oder die versteckten Daten rekonstruiert werde. Daher ist es wichtig, die Angriffsvektoren und -risiken zu kennen. Auf Basis dessen kann dann ein Schutzkonzept erarbeitet und das Programm optimal geschützt werden.

2.3. Anwendungsbereiche

Soll der Algorithmus geschützt werden, spricht man von **Code Obfuscation**. Dabei wird darauf geachtet, dass ein im Code implementierter Algorithmus derart verschleiert wird, dass dieser nur schwer rekonstruiert und verstanden werden kann. Dies erfolgt durch Änderung der Abfolge von Befehlen, um die wahre Abfolge zu verschleiern.
Bei der **Daten Obfuscation** müssen im Code abgelegte Daten derart verfremdet werden, dass eine Rekonstruktion nur schwierig erfolgen kann.
Als dritte Form kann die **Quellcode Obfuscation** genannt werden. Dabei wird der Quellcode schon derart verfremdet, dass dieser schwer lesbar ist. Dies kommt vor allem bei Skriptsprachen zum Einsatz.

2.4. Code Obfuscation

Bei der Code Obfuscation wird versucht, die eigentliche Funktion des Programms zu verschleiern. Dabei kann es sich um einzelne Teile wie zum Beispiel bestimmte Algorithmen oder einen Lizenzcheck handeln oder es wird das ganze Programm verschleiert. Bei der Code Obfuscation wird der Programmablauf durch Einfügen zusätzlicher Verzweigungen und Schleifen verfremdet. Mathematische Operationen können durch äquivalente Ausdrücke ersetzt werden. Es kann Code eingefügt werden, der nie durchlaufen wird, jedoch Reverseengineering-Ressourcen bindet. All diese Techniken führen zu einem komplexen Codeablauf, der schwieriger zu verstehen ist als der ursprüngliche. In Abbildung 2.1 ist ein Beispielprogramm in der nicht verschleierten Form dargestellt. Im Gegensatz dazu zeigt Abbildung 2.2 die verschleierte Variante vom in Abbildung 2.1 gezeigten Programm.

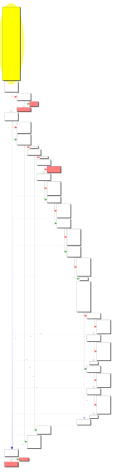

Abbildung 2.1.: Beispielprogramm in der nicht verschleierten Form

Abbildung 2.2.: Beispielprogramm in der Code verschleierten Form

2.5. Daten Obfuscation

Es ist manchmal notwendig, Daten in einem Programm zu verstecken. Dabei kann es sich zum Beispiel um kryptografische Schlüssel, wie beim Content Scramble System der DVD, oder um reinen Text handeln. Damit die Daten beim Analysieren nicht offensichtlich ausgelesen werden können, werden diese in einer anderen Ausprägung abgelegt. Dies erfolgt meist durch eine mehr oder weniger komplexe Verschlüsselung. Oft reicht eine einfache XOR-Verschlüsselung, um die Zeichenfolge als solche nicht zu erkennen. Die Listings 2.1 und 2.2 zeigen Zwischencode generiert aus einem C-Beispielprogramm „Hello World“. In Listing 2.1 ist die Zeichenkette „Hello World“ eindeutig erkennbar und ist auch in dieser Form in der binären Datei sichtbar. Listing 2.2 zeigt die Daten verschleierte Version des Zwischencodes. In diesem ist die Zeichenkette nicht offensichtlich lesbar. Auch in der binären Datei ist die Zeichenkette nicht sichtbar.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
```

Listing 2.1: Auszug aus dem Datenbereich vom Zwischencode von einem C-Programm „Hello World“

```
@EncryptedString = private global [14 x i8] c"\18\B4\EE\AF\F3\C7\8FbGm\9B1‘K"
```

Listing 2.2: Auszug aus dem Datenbereich vom Zwischencode von einem C-Programm „Hello World“ welches Daten verschleiert wurde

2.6. Quellcode Obfuscation

Die Quellcodeverschleierung kommt vor allem dort zum Einsatz, wo Quellcode bis zum Endkunden gelangt. Dies ist vor allem bei Skriptsprachen wie zum Beispiel „javascript“ der Fall. Hier wird Quellcode bis in den Browser des Endanwenders übertragen und dieser hat somit Zugriff darauf. Bei der Quellcodeverschleierung wird versucht, es zu erschweren den Quellcode zu lesen. Dies erfolgt zum Beispiel mit Umcodierung in hexadezimale Darstellung, Umformatierung durch Entfernen der Zeilentrenner oder durch Techniken, die auch aus der Code Obfuscation oder Daten Obfuscation bekannt sind [14]. Eine weitere bekannte Anwendung sind Bewerbe zur Verschreibung von Quellcode. Dabei geht es darum, wer die beste Verschleierung von einfachen Programmen wie zum Beispiel „Hello World“ umsetzen kann [15]. Listing 2.3 zeigt ein einfaches Programm wie „Hello World“ in der Programmiersprache C umgesetzt ist. Die darauffolgenden Listings 2.4 und 2.5 zeigen die Darstellung des einfachen Beispiels in der Ausprägung der Code und Daten Obfuscation. Bei der Daten Obfuscation ist der String „Hello World“ als solches nicht mehr erkennbar. Hingegen ist im zweiten Beispiel in Listing 2.5 der String als solches erkennbar, jedoch kann keine Aussage über die Funktion getroffen werden.

```
#include<stdio.h>
int main() {
  printf("Hello World!\n");
  return 0;
}
```

Listing 2.3: Einfaches Programmbeipiel für "Hello World“ in der Programmiersprache C

```
#include <stdio.h>
int main() {
```

```
float b[] = {1.1431391224375825e+27, 6.6578486920496456e+28, 7.7392930965627798e-19,
    3.2512161851627752e-9};
puts(b);
return 0;
}
```

Listing 2.4: Eine Version eines datenverlscheierten "Hello World“ in der Programmiersprache C, dabei ist der String "Hello World“ als solches nicht mehr erkennbar

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Listing 2.5: Eine Version eines codeverschleierten "Hello World“ in der Programmiersprache C, hier ist die Funktion des Programms nicht auf den ersten Blick ersichtlich, bei dem Beispiel handelt es sich um einen anonym eingereichten Code des Obfuscation C Code Contest von 1984

2.7. Tools

Dieser Abschnitt bietet eine Übersicht der für die Programmiersprachen C und C++ verfügbaren Obfuscation-Tools. Am Markt gibt es kommerzielle und einige freie Tools. Generell ist aber zu erkennen, dass es für die betrachteten Programmiersprachen nur mehr wenige freie und aktive Projekte gibt. Die folgende Auflistung soll zeigen welche Tools verfügbar sind und welche davon noch aktiv entwickelt werden.

2.7.1. CXX-Obfus

Bei CXX-Obfus handelt es sich um ein kommerzielles Programm, das für die Plattformen Windows, MacOS und Linux verfügbar ist [16]. Es verschleiert den Quellcode von C und C++ Programmen, welche anschließend kompiliert werden können. Folgende Methoden der Verschleierung werden unterstützt:

- Zeichenketten in Hexadezimaldarstellung
- Integer Zerlegung
- Entfernen von Leerzeichen und Zeilenumbrüchen
- Umbenennen von symbolischen Namen
- Umbenennen von numerischen Konstanten
- Umbenennen von Dateinamen und Verzeichnissen
- Entfernung von Kommentaren

Aktuell ist die Version 4.7 zum Download verfügbar. Diese Version wurde am 12. Dezember 2017 veröffentlicht. Das Tool selber gibt es seit 2004. In den Referenzen scheinen Firmen wie Cisco, DELL, Bosch, Broadcom, Ericsson, Siemens und Citrix auf.

2.7.2. COBF

Bei COBF handelt es sich um einen freien C/C++ Quellcode Obfuscator [17]. Die letzte verfügbare Version ist V1.06 von 2006. Dieses Tool verschleiert den Quellcode von C und C++ Programmen. Folgende Methoden werden genannt:

- Umbenennen von Funktionen
- Umbenennen von Variablen
- Umbenennen von Dateinamen und Verzeichnissen
- Entfernen von Leerzeichen und Zeilenumbrüchen
- Entfernung von Kommentaren

Als Plattformen werden Windows und Linux unterstützt.

2.7.3. C Source Code Obfuscator

C Source Code Obfuscator [18] ist ein kommerzielles Tool und, so wie der Name sagt, verschleiert es den Quellcode. Es gibt keine Information darüber ob das Tool noch weiterentwickelt wird. Es werden folgende Features angegeben:

- Umbenennen von Funktionen
- Umbenennen von Variablen
- Entfernen von Leerzeichen und Zeilenumbrüchen
- Entfernung von Kommentaren

2.7.4. Snob

Snob [19] steht für **s**imple **n**ame **ob**fuscator. Dabei handelt es sich um einen Quellcode Obfuscator, der nach regulären Ausdrücken sucht und diese ersetzt. Damit ist es ein universelles und frei konfigurierbares Tool. Die letzte Version stammt von 2010.

2.7.5. Strings Obfuscation System

Beim String Obfuscation System [20] handelt es sich um einen reinen Zeichenketten Obfuscator. Zeichenketten werden mittels AES-256 verschlüsselt. Für jeden String wird ein eigener Schlüssel erzeugt.

Das Tool stellt die C Funktionen zum ver- und entschlüsseln bereit. Letzte Aktivitäten bei dem Projekt sind von 2013.

2.7.6. Obfuscator

Beim Projekt Obfuscator [21] handelt es ich um einen webbasierten Obfuscator. Auf der Webseite kann C und C++ Code einfach und schnell in verschleierten Sourcecode verwandelt werden. Leider sind keine näheren Informationen darüber bekannt welche Techniken dabei angewendet werden.

2.7.7. StarForce C++ Obfuscator

Das komerzielle Tool StarForce C++ Obfuscator [22] wird von Firmen wie Corel, Spotify, General Electrics und netmarble eingesetzt. Dabei handet es sich nicht um einen Quellcode Obfuscator sondern es wird ein Zwischencode, ein virtueller Maschinencode, erzeugt. Das Tool unterstützt mehr als 30 verschiedene Obfuscation Methoden. Die meisten können einzeln aktiviert werden. Die wichtigsten Methoden sind:

- Erzeugung aus C++ Code virtuellen Maschinencode
- Verschlüsselung von Zeichenketten und Arrays
- Einführung von Zustandsmaschinen
- Einfügen von Dummy Code
- Zusammenfügen von Code Bereichen

StarForce C++ Obfuscator ist für Windows, MacOS und Linux verfügbar.

2.7.8. Cloakware Software Protection

Cloakware Software Protection [23] ist ein kommerzielles Tool, das für die Plattformen Windows, MacOS und Linux verfügbar ist. Es unterstützt die Programmiersprachen C, C++, Swift und Javascript. Es werden auch die mobilen Plattformen Android und iOS unterstützt. Das Tool besteht aus 3 Komponenten, welche sich gegenseitig ergänzen:

- Transformator / Obfuscator
- WhiteBox Verschlüsselung
- Integrität Überprüfung

Der Transformator ist das Kernstück der Software Protection Suite. Dieser führt die eigentlichen Verschleierungen durch. Dabei kann dieser auf die Verschlüsselungskomponente zurückgreifen, um sensible Daten wie kryptographische Schlüssel durch Standardverschlüsselung (AES, ECC oder RSA) zu schützen. Das Integritätsmodul erkennt Veränderungen an der Software und zeigt diese auf.

2.7.9. Loco

Loco [24] ist ein Tool aus dem Forschungsbereich, das eine grafische Oberfläche besitzt und interaktiv ist. Damit können Auswirkungen von Obfuscation evaluiert werden. Das Tool kann den Programmfluss verändern. Die Basis für Loco bildet Diablo für die Obfuscation und LANCET für den grafischen Teil. Loco wurde 2006 mit der Publikation [24] veröffentlicht, seither wurden keine Updates vorgenommen.

2.7.10. Obfuscator-LLVM

Obfuscator-LLVM ist 2010 von der Fachhochschule HEIG-VD ins Leben gerufen worden. Das Projekt basiert auf der Compiler-Suite LLVM. Diese Suite besitzt zentrale Optimierungsmodule. Von dem Projekt wurden zusätzlich Obfuscatormodule entwickelt, die statt den Optimierungsmodulen verwendet werden können. Die daraus entstandene Obfuscator-Suite unterstützt alle auch von LLVM unterstützten Programmiersprachen, da die Optimierungsmodule und somit auch die Obfuscatormodule auf Zwischencode arbeiten. Die letzte Änderung an diesem Projekt stammt von 2018.

2.7.11. Hikari

Da die letzte Änderung von Obfuscator-LLVM von 2018 ist und die Suite auf dem veralteten LLVM 4 basiert, hat Zhang das Projekt Hikari [25] initiiert. Hikari gibt es seit 2018 und wird aktuell entwickelt. Im Jänner 2019 wurde Hikari für LLVM 7 veröffentlicht. Seit März 2019 ist auch Hikari basierend auf LLVM 8 verfügbar. Damit ist Hikari aktuell das aktivste freie Obfuscator-Projekt.

3. Obfuscation-Techniken

Dieses Kapitel bietet eine Übersicht der bekannten Obfuscation-Techniken. Nach Collberg et al. [26] können diese Techniken grob in 4 Gruppen unterteilt werden:

- Layout Transformation
- Kontroll Transformation
- Daten Transformation
- Preventive Transformation

Die ersten drei Gruppen werden im Folgenden genauer beschrieben. Zu jeder der drei Gruppen werden die gebräuchlichsten Techniken und Methoden zur Obfuscation gezeigt. Diese Aufzählung erhebt keinen Anspruch auf Vollständigkeit.

3.1. Layout Transformation

Layout Transformationen [27] können im Quellcode vorgenommen werden und sind relativ einfach umzusetzen. Der Vorteil dieser Transformationen ist, dass sie in der Regel unumkehrbar sind. Das heißt sie führen zu einem Informationsverlust.

3.1.1. Umbenennen von Variablen und Funktionen

Eine der einfachsten Techniken zur Verschleierung von Code ist das Umbenennen von Variablen. Durch die Verwendung von Variablennamen [7, S. 203ff], die keinen Rückschluss auf den Inhalt zulassen, wird das Analysieren von Code erschwert. Dabei kann die Umbenennung bereits im Quellcode erfolgen. Aber auch eine Veränderung der Namen im Compiler ist möglich. Ein weiterer Schritt ist die Umbenennung der Namen von Funktionen und Methoden. Dabei muss aber darauf geachtet werden, ob es sich um lokale oder externe Funktionen handelt. Diese Technik wird auch „Identifier Renaming“ [7, S. 203ff] oder „Name Scrambling“ [28] genannt.

3.1.2. Verändern des Formats

Schon das Entfernen von Leerzeichen und Zeilenumbrüchen trägt dazu bei, dass ein Quellcode unleserlich wird. Diese Technik alleine wird noch keine Analyse verhindern, aber in Kombination mit anderen Techniken trägt sie dazu bei, die Analyse zu erschwerten.

3.1.3. Entfernen von Kommentaren

Gute Kommentare sind oft eine extreme Hilfestellung, um komplexen Quellcode zu verstehen. Durch die Entfernung der Kommentare wird diese zusätzliche Information unwiderruflich gelöscht und kann nicht zur Analyse verwendet werden.

3.2. Kontroll Transformation

Kontroll Transformationen [27] werden in der Regel im Compiler umgesetzt. Dabei wird der Kontrollfluss des Programms derart verändert, dass der interne Ablauf viel komplexer erscheint als er in Wahrheit ist. Nach außen hin sind die funktionalen Eigenschaften des Programms gleich. Die nicht-funktionalen Eigenschaften, wie zum Beispiel die Laufzeit der Verarbeitung, können abweichen, da sich durch die künstlich geschaffene interne Komplexität die Ausführungszeit verlängert.

3.2.1. Einfügen von Code der nie oder ohne Funktion durchlaufen wird

Diese Technik ist auch bekannt as „Inserting Dead or Irrelevant Code“ [26,28]. Der Begriff „Toter Code“ bezieht sich auf Codeblöcke die durch den Kontrollfluss des Programms nie durchlaufen werden. Diese Blöcke können eine zufällige Ansammlung oder ein vorbereiteter Satz von Befehlen sein, die bei der Analyse zeitaufwendig zu analysieren sind. Je nachdem wie viel Code eingefügt wird, ist auch ein Anstieg der Dateigröße merkbar. Irrelevanter Code sind Befehle, die zwar durchlaufen werden, aber nichts zur Funktion eines Programms beitragen. Der einfachste dieser Befehle ist der „No Operation“ (NOP) - Befehl. Jedoch können auch Befehlssequenzen wie eine Addition, einige Zeilen später gefolgt von einer Subtraktion der gleichen Zahl, verwendet werden, um zusätzliche Komplexität in den Code zu bringen.

3.2.2. Umordnung

Die Umordnung wird auch „Reordering“ [26,28] genannt. Programmierer und Programmiererinnen platzieren im Allgemeinen Befehle, die zusammen gehören, nahe beieinander. Das „Reordering“ beruht darauf, dass Befehle, die voneinander unabhängig sind, im Code beliebig verteilt werden können und damit der Zusammenhang nicht leicht erkennbar ist.

3.2.3. Undurchsichtige Ausdrücke

Die undurchsichtigen Ausdrücke werden in der Fachliteratur auch „Opaque Predicates“ [28] oder „Opaque Expressions“ [26] genannt. Der Begriff Prädikat kommt aus der Prädikatenlogik und beschreibt den Teil einer atomaren Aussage, der wahrheitsfunktional ist [29]. Vereinfacht gesagt, werden Verzweigungen mit logischen Ausdrücken in den Code eingefügt, welche immer im gleichen Zweig durchlaufen werden. Welchen Zweig das Programm durchläuft, wird ist zum Zeitpunkt der Obfuscation festgelegt. Durch statische Analyse ist es schwer festzustellen, welche Zweige nie durchlaufen werden. Die Zweige die nicht durchlaufen werden, enthalten somit „Dead Code“, siehe Abschnitt 3.2.1. Als Weiterentwicklung gibt es auch „Dynamic Opaque Predicates“ [30], welche versuchen, nicht vorhersehbar zu sein. „Opaque Predicates“ bilden die Grundlage für „Bogus Control Flow".

3.2.4. Kontrollfluss verschleiern

Basierend auf den „Opaque Predicates“ (siehe Abschnitt 3.2.3), werden zusätzliche Verzweigungen in den Code eingefügt. Dies wird auch als „Bogus Control Flow“ [26] bezeichnet.

3.2.5. Kontrollfluss Verflachung

Diese Methode wird auch „Control Flow Flattening“ [**?**] genannt und bringt die Basisblöcke in einen flachen Kontrollfluss. Dies geschieht zum Beispiel mit einer Switch-Anweisung zu Beginn. Die Basisblöcke des Programms sind in den einzelnen Zweigen abgebildet. Eine zusätzliche Logik steuert die Switch-Anweisung, um die Basisblöcke in der richtigen Reihenfolge zu durchlaufen. Bei der statischen Analyse ist die Reihenfolge in der die Basisblöcke durchlaufen werden schwierig feststellbar.

3.2.6. Nicht reduzierbarer Kontrollfluss

Normalerweise besteht der Kontrollfluss aus if-,for-, while-, case-Anweisungen. Diese ergeben einen reduzierbaren Kontrollfluss und sind effizent zu analysieren. Eine Sonderform des „Bogus Control Flow“ ist der nicht reduzierbare Kontrollfluss [**?**] auch „Irreducible Flow graph“ genannt. Dabei wird zum Beispiel durch „Opaque Predicates“ mittels der Befehle GOTO oder LABEL inmitten einer Schleife gesprungen. Diese Konstrukte entsprechen keinem reduzierbaren Kontrollfluss mehr.

3.2.7. Schleifentransformation

Schleifentransformationen [28] wurden entwickelt, um die Effizienz der Ausführung im Bezug auf die Cache-Nutzung zu optimieren. Die daraus entstandenen Konstrukte eignen sich auch zur Obfuscation.

So werden Schleifen zum Beispiel in eine Schleife mit inneren Schleifen zerlegt. Dies optimiert unter Umständen die Effizienz, erhöht aber die Komplexität bei der Analyse.

3.2.8. Befehlsersetzung

Bei der Befehlsersetzung, auch „Instructions Substitution“ genannt, werden vor allem mathematische Ausdrücke durch äquivalente, aber komplexere Ausdrücke ersetzt.

3.3. Daten Transformation

Müssen geheime Daten wie zum Beispiel kryptograpfische Schlüssel in einer Binärdatei ausgeliefert werden, sollen diese vor einer Analyse und anschließenden Extraktion geschützt werden. Dabei werden die Werte von Variablen bei ihrer Zuweisung verschleiert. Im Folgenden ist ein Auszug der vorhandenen Techniken angeführt:

3.3.1. Umkodierung

Bei dieser Technik werden die initialen Zuweisungen von Variablen verschleiert. Die initialen Werte werden in eine Repräsentation gebracht, um es zu erschweren den Wert zu einfach zu lesen [7, S. 258]. Dies kann zum Beispiel eine Zeichenkette sein, die durch ein einfaches Verschlüsselungsverfahren wie ROT13 [31] verschleiert wird. Wird Umkodierung angewendet, ist es notwendig 2 Funktionen für die Obfuscation bereitzustellen. Um die Zeichenkette zu verschleiern, ist eine Funktion zum Encodieren erforderlich. Bei der Ausführung des Programms, indem sich die Zeichenkette befindet, wird eine weitere Funktion zum Dekodieren benötigt, um auf den ursprünglichen Wert zu kommen.

3.3.2. Teilen und Zusammenfügen von Variablen

Bei boolschen Variablen ist es schwierig, eine Umkodierung vorzunehmen. Hier wird ein anderer Weg beschritten. Logische Variablen können in mehrere Sub-Variablen aufgeteilt werden [7, S. 268], welche an unterschiedlichen Orten im Code abgelegt werden. Der eigentliche boolsche Wert ergibt sich aus der Verknüpfung der einzelnen Sub-Variablen.

3.3.3. Verschleiern von Daten-Arrays

Arrays bieten vielfältige Möglichkeiten, die Daten zu verschleiern. Eine einfache Möglichkeit ist es, die Daten im Array umzusortieren. Für die Dekodierung wird eine Funktion benötigt, welche die Zuordnung zwischen Index und Daten wiederherstellt. Wie auch bei primitiven Datentypen, besteht die Möglichkeit

das Array zu zerteilen und in getrennten Bereichen abzuspeichern. Eine weitere Verschleierungstechnik ist das Falten. Dabei werden zum Beispiel eindimensionale Arrays in zwei- oder mehrdimensionale Felder transformiert.

4. LLVM Compiler Infrastruktur

Früher ist LLVM als „low level Virtual Machine“ bezeichnet worden. Heute stellt dies aber keine Abkürzung dar, sondern ist der volle Name des Projekts [32]. LLVM ist von der Universität von Illinois als Forschungsprojekt begonnen worden. Das Ziel ist es, einen modernen und SSA basierten Compiler zu entwickeln, der auch verschiedene Programmiersprachen unterstützt. SSA ist die Abkürzung für „static single assignment“ und beschreibt, wie im Zwischencode mit Variablen umgegangen wird [33]. Bei SSA wird einer Variable nur einmal ein Wert zugewiesen. Dadurch können Abhängigkeiten von Daten und Befehlen besser dargestellt werden. Dies erleichtert die Optimierung des Codes. LLVM besteht wie die meisten modernen Kompiler aus 3 Stufen [34]. Die erste Stufe, das Frontend, liest den Quellcode ein, überprüft diesen auf dessen Syntax und erstellt einen Abstract Syntax Tree (AST). Dieser abstrakte Syntax Baum wird in die Zwischensprache LLVM IR (Intermediate Response) übersetzt. LLVM IR ist der Input für die 2. Stufe, den Optimierer. Dieser optimiert den IR Code und stellt den optimierten Code dem Backend zur Verfügung. Das Backend übernimmt die Generierung des Maschinencodes für die jeweilige Plattform. Diese 3 Stufen sind in der Abbildung 4.1 dargestellt.

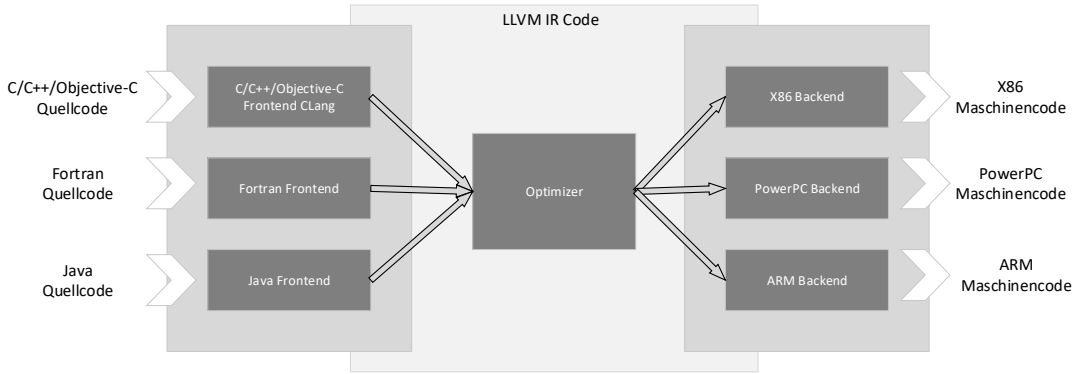

Abbildung 4.1.: 3 Stufen Compiler

Der modulare Aufbau ermöglicht es, Frontends unabhängig vom Optimizer und den Backends zu entwickeln. Ebenso von Vorteil ist es, dass die einzelnen Optimierungsmodule in eine Sprache, der LLVM IR,

entwickelt werden können. Im Gegensatz zu den monolithischen Compilern wie zum Beispiel GCC, ist es durch die modulare Architektur von LLVM möglich, die Backends als Just-in-Time (JIT) Compiler in eine Applikation zu integrieren.

4.1. LLVM IR

Die Zwischensprache LLVM IR ist mit dem Ziel entwickelt worden, 3 verschiedene Bereiche abzudecken:

- In-memory Zwischensprache
- On-disk Bytecode
- Menschen lesbarer Assemblercode

Bei der Entwicklung von LLVM IR ist darauf geachtet worden, dass die Zwischensprache auf der einen Seite leichtgewichtig und low-level ist, auf der anderen Seite ist sie strukturiert und erweiterbar. Eine Referenz zu dieser Sprache ist im LLVM Language Reference Manual [35] zu finden.

4.2. Frontends

Aktuell gibt es eine große Auswahl an Frontends. Diese unterstützen nicht nur prozedurale Programmiersprachen wie C, Delphi oder Fortran sondern auch objektorientierte Sprachen wie C++, C# Objective C++, GO und Java oder funktionale Programmiersprachen wie Haskell. Die Unterstützung der einzelnen Sprachen ist in verschiedenen Frontends implementiert. Ein wichtiges Frontend ist Clang.

4.2.1. Clang

Clang ist ein Frontend der LLVM Compiler Infrastruktur, welches mit dem monolithischen Compiler GCC vergleichbar ist. Es unterstützt die Programmiersprachen der C Familie, wie C, C++, Objective C/C++, OPenCL, CUDA und Rendersript [36]. Die Merkmale von Clang sind:

- Kompiliert schneller als GCC und verbraucht weniger Speicher (laut [36])
- Aussagekräftigere Fehlermeldungen als GCC
- GCC kompatibel
- Library basierter Ansatz
- LLVM BSD Lizenz

Clang wandelt die oben genannten Programmiersprachen in den IR Zwischencode um. der in weiterer Folge durch die Optimizerstufe optimiert wird und dann an das Backend weitergereicht wird. Im Gegensatz zu anderen Compilern unterstützt Clang umfangreiche und genaue Analysemethoden, um eine

Fehlersuche zu erleichtern.

4.3. Optimizer

Eine große Stärke der LLVM Compiler Infrastruktur ist die Optimierung. Diese ist der zentrale Baustein von LLVM. Der Optimizer nimmt IR Code vom Frontend entgegen, analysiert und transformiert diesen in verschiedenen Durchläufen (Passes). Analysedurchläufe sammeln Informationen, welche als Debug-Informationen oder zur Visualisierung genutzt werden. Transformationsdurchläufe sind dazu vorgesehen, den IR Code zu verändern. Als dritte Kategorie gibt es Utility-Passes, welchen aber weniger Bedeutung zukommt.

4.3.1. Analyse-Passes

Analyse-Passes verändern den IR Zwischencode nicht. Die Module sammeln Informationen, welche für Statistiken, Visualisierung oder als Input für folgende Transformation-Passes genutzt werden können. In dieser Kategorie gibt es aktuell ca. 40 Module.

4.3.2. Transformation-Passes

Transformationen können auf Befehle, Basisblöcke, Funktionen oder Module im IR Zwischencode angewendet werden. Vorgeschaltete Analysedurchläufe können Informationen an Transformationsdurchläufe übergeben. Jede Transformation muss gültigen und validen Code erzeugen und kann nicht von vorher angewendeten Transformation-Passes abhängig sein. In der Kategorie Transformation-Passes gibt es momentan ca. 60 Module, einige Vertreter sind zum Beispiel:

- Dead Code Elimination
- Global Variable Optimizer
- MemCpy Optimization
- Merge Functions
- Scalar Replacement of Aggregates

4.3.3. PassManagerBuilder

Der PassManager kümmert sich im Zuge der Optimierung um die Verwaltung der Ergebnisse der Analyse-Passes, die Speicherverwaltung und der zugehörigen Passe [37].

Backends

Die 3. Stufe, das Backend, übersetzt den vom Optimizer verarbeiteten IR Zwischencode für die einzelnen Prozessor-Architekturen. LLVM unterstützt ein breites Spektrum an Architekturen, wie zum Beispiel: x86, amd64, PowerPC, ARM, SPARC, MIPS, ALPHA, usw. Zusätzlich gibt es Projekte, welche eigene Backends für LLVM entwickeln. Zu erwähnen wäre hierbei das Mono-Projekt. Mono hat ein Backend für LLVM [38] entwickelt, um den Mono-Code, welcher normalerweise mittels Just-in-Time Compiler läuft, in nativen Code zu verwandeln.

5. Projekt „Obfuscator-LLVM“

Das Projekt Obfuscator-LLVM wird 2010 von der Fachochschule HEIG-VD in der Schweiz ins Leben gerufen [39]. Das Ziel ist es eine auf dem LLVM basierende Compiler Infrastruktur zu erstellen, mit der es möglich ist, die Sicherheit von Software im Hinblick auf Manipulation und Reverse-Engineering zu erhöhen. Dazu werden verschiedene Code-Transformationen in die LLVM-Compiler Basisinfrastruktur implementiert. Im Jahr 2015 gibt es eine frei und eine kommerzielle Version von der Obfuscator-LLVM Suite. In der freien Open Source Variante sind die meisten Verschleierungstechniken implementiert [40]. In der kommerziellen sind noch weitere Techniken, wie zum Beispiel „Function Merging“ oder „Tamper Proofing“ vorhanden. Zum jetzigen Zeitpunkt ist die kommerzielle Version nicht mehr verfügbar und das Projekt wird nicht mehr weiterentwickelt. In der letzten Version setzt Obfuscator-LLVM auf der LLVM-Suite in der Version 4 auf, wobei LLVM aktuell in der Version 7 verfügbar ist. Mit dem Obfuscator-LLVM ist es möglich, alle vom LLVM-Frontend unterstützen Programmiersprachen zu verschleiern. Bei der Verwendung von Obfuscator-LLVM werden auch alle Architekturen von der LLVM Suite unterstützt. Normalerweise sind Compiler deterministisch, das heißt jeder Durchlauf des Compilers mit ein und demselben Quellcode liefert auch das selbe Ergebnis. Die Verschleierungsmodule von Obfuscator-LLVM bringen Diversität in den erzeugten Binärcode, sodass jede implementierte Verschleierungstechnik ein bestimmtes Maß an Zufälligkeit in den Binärcode bringt. Diese wird bei den einzelnen Techniken im Anschluss beschrieben. Für die Zufälligkeit wurde ein einfacher, kryptographisch sicherer, Pseudo-Zufallsgenerator (PRNG) implementiert. Dieser basiert auf einem AES128 Algorithmus, welcher nach dem NIST-Standard umgesetzt wird. Im Folgenden wird auf die Installation von Obfuscator-LLVM und auf die einzelnen Verschleierungstechniken näher eingegangen. Die ersten 3 Techniken sind in der Open Source Variante frei zugänglich, die weiteren sind nur in der kommerziellen Version verfügbar.

5.1. Installation

Die Installation ist teilweise nach der Anleitung [41] der Projektseite LLVM Obfuscator durchgeführt. Leider funktioniert die beschriebene Installation nicht problemlos. In Listing 5.1 ist die modifizierte und funktionierende Anleitung angeführt. Obfuscator-LLVM lässt sich nach dieser Anleitung auf Debian 9

kompilieren.

```
git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git
mkdir build
cd build
cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release -DLLVM_INCLUDE_TESTS=OFF ../
    obfuscator/
make -j7
```

Listing 5.1: Modifizierte Installationsanleitung für LLVM Obfuscator

Als Ergebnis stehen die Compiler Binaries, welche für die weitere Verwendung von Obfuscator-LLVM relevant sind, im Verzeichnis build/bin zur Verfügung.

5.2. Umgesetzte Verschleierungstechniken

5.2.1. Instruction Substitution

Das Ersetzen von einzelnen Befehlen ist die einfachste Methode, um einen Code zu verschleiern. Bei der hier eingesetzten Technik werden arithmetische Befehle wie „ADD“ und „SUB“, aber auch binäre Operationen wie „AND“, „OR“ und „XOR“ durch äquivalente aber komplexere Befehle ersetzt. Für die beiden arithmetischen Befehle gibt es mehrere äquivalente Alternativen, welche über den vorher erwähnten Zufallsgenerator ausgewählt werden. In Tabelle 5.1 sind die äquivalenten Alternativen zum Befehl „ADD“, SUB“ und die Binäroperationen dargestellt. Der Einsatz von der Substitution wird über 2 Parameter gesteuert. Die Befehlsersetzung kann mittels `-mllvm -sub` aktiviert werden. Der zusätzliche Parameter `-sub_loop=n` gibt an, dass die Substitution n-mal auf die Funktion angewendet werden soll. Der Standardwert für `n` ist 1. Eine Zusammenfassung ist in Listing 5.2 zu finden. Diese Art der Obfuscation ist sehr einfach und bietet keinen ausreichenden Schutz gegen Reverse-Engeneering. Die Technik kann recht einfach durch Optimierung des Compilers rückgängig gemacht werden. Das vorrangige Ziel dieser Technik ist es, Diversität in den Maschinencode zu bekommen.

```
C(XX)FLAGS : -mllvm -sub -sub_loop=n (default n=1)
```

Listing 5.2: CFLAGS / CXXFLAGS der Befehlsersetzung

Test der Funktion

Für den Test der Verschleierungsfunktion „Intruction Substitiution“ wird ein Beispielprogramm für arithmetische und binäre Operationen erstellt. Diese ist in Listing 5.3 abgebildet. Dabei sind alle Funktionen, welche von der Obfuscation-Technik unterstützt werden, vorhanden.

ursprünglicher Befehl	äquivalente Varianten
a = b + c	a = b - (- c)
	a = - (- b + (- c))
	r = RND(); a = b + r; a += c; a -= r;
	r = RND(); a = b - r; a += c; a += r;
a = b - c	a = b + (- c)
	r = RND(); a = b + r; a -= c; a -= r;
	r = RND(); a = b - r; a -= c; a += r;
$a = b \wedge c$	$a = (b \oplus \overline{c}) \wedge b$
$a = b \vee c$	$a = (b \wedge c) \vee (b \oplus c)$
$a = b \oplus c$	$a = (\overline{b} \wedge c) \vee (b \wedge \overline{c})$

Tabelle 5.1.: Äquivalenzoperationen für die arithmetischen Funktionen addieren und subtrahieren, sowie für die binären Operationen UND, ODER und exklusives ODER. Bei den arithmetischen Operationen wird zufällig keine Variante ausgewählt.

```c
#include<stdio.h>
int main(){
  int v = 1;
  int i;
  int j;
  int a;
  int b;
  int c;
  i = 10 + v;
  j = 10 - v;
  a = 0 & v;
  b = 0 | v;
  c = 0 ^ v;
  return 0;
}
```

Listing 5.3: Testprogramm für arithmetische Funktionen

Das Programm wird mit den Befehlen aus Listing 5.4 übersetzt, um jeweils eine ausführbare und eine Version mit IR Zwischencode zu erhalten. Die Dateien mit der Endung .ll enthalten den IR Code. Dateien mit dem Wort „llvm“ im Dateinamen sind mit dem Clang ohne Verschleierung übersetzt worden. Dateien mit dem Wort „llvmObf“ im Dateinamen sind mit „Intruction Substitiution“ kompiliert.

```
#LLVM Applikation
clang-4.0 arithmetic.c -o arithmetic.llvm
#LLVM IR Zwischencode
clang-4.0 arithmetic.c -emit-llvm -S -o arithmetic.llvm.ll

#"Intruction Substitiution" verscheierte Applikation
clang-4.0 arithmetic.c -mllvm -sub -o arithmetic.llvmObf
#"Intruction Substitiution" verscheierter IR Zwischencode
clang-4.0 arithmetic.c -emit-llvm -S -mllvm -sub -o arithmetic.llvmObf.ll
```

Listing 5.4: Befehle zum Kompilieren ohne und mit „Function Wrapper“

Listing 5.5 zeigt den erzeugten LLVM IR Code ohne Verschleierung. Dabei sind die Blöcke mit den einzelnen arithmetischen Funktionen kommentiert.

```
; Function Attrs: noinline nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4 ;v
  %3 = alloca i32, align 4 ;i
  %4 = alloca i32, align 4 ;j
  %5 = alloca i32, align 4 ;a
  %6 = alloca i32, align 4 ;b
  %7 = alloca i32, align 4 ;c

  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4 ;initialize v=1;
  ; i = 10 + v
  %8 = load i32, i32* %2, align 4
  %9 = add nsw i32 10, %8
  store i32 %9, i32* %3, align 4
  ; j = 10 - v
  %10 = load i32, i32* %2, align 4
  %11 = sub nsw i32 10, %10
  store i32 %11, i32* %4, align 4
  ; a = 0 & v
  %12 = load i32, i32* %2, align 4
  %13 = and i32 0, %12
  store i32 %13, i32* %5, align 4
  ; b = 0 | v
  %14 = load i32, i32* %2, align 4
  %15 = or i32 0, %14
  store i32 %15, i32* %6, align 4
  ; c = 0 xor v
  %16 = load i32, i32* %2, align 4
  %17 = xor i32 0, %16
  store i32 %17, i32* %7, align 4
  ret i32 0
}
```

Listing 5.5: LLVM IR

In Listing 5.6 ist eine verschleierte Variante in der IR Zwischencode Darstellung zu sehen. Die Blöcke

der einzelnen arithmetischen Funktionen sind, wie bereits in Listing 5.5, durch Kommentare kenntlich gemacht worden. Bei mehrmaligem Ausführen werden die aus der Tabelle 5.1 zur Verfügung stehenden Funktionen zufällig ausgewählt und eingesetzt. Somit ergibt sich eine Diversität im Maschinencode und Substitutionen können nicht so leicht mit Patterns erkannt werden.

```
; Function Attrs: noinline nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4 ;v
  %3 = alloca i32, align 4 ;i
  %4 = alloca i32, align 4 ;j
  %5 = alloca i32, align 4 ;a
  %6 = alloca i32, align 4 ;b
  %7 = alloca i32, align 4 ;c

  store i32 0, i32* %1, align 4
  store i32 1, i32* %2, align 4 ; v=1
  ; i = 10 + v
  %8 = load i32, i32* %2, align 4
  %9 = add i32 10, -160417045
  %10 = add i32 %9, %8
  %11 = sub i32 %10, -160417045
  %12 = add nsw i32 10, %8
  store i32 %11, i32* %3, align 4
  ; j = 10 - v
  %13 = load i32, i32* %2, align 4
  %14 = sub i32 10, -941050624
  %15 = sub i32 %14, %13
  %16 = add i32 %15, -941050624
  %17 = sub nsw i32 10, %13
  store i32 %16, i32* %4, align 4
  ; a = 0 & v
  %18 = load i32, i32* %2, align 4
  %19 = xor i32 0, -1
  %20 = xor i32 %18, -1
  %21 = xor i32 -815727026, -1
  %22 = or i32 %19, %20
  %23 = or i32 -815727026, %21
  %24 = xor i32 %22, -1
  %25 = and i32 %24, %23
  %26 = and i32 0, %18
  store i32 %25, i32* %5, align 4
  ; b = 0 | v
  %27 = load i32, i32* %2, align 4
  %28 = and i32 0, %27
  %29 = xor i32 0, %27
  %30 = or i32 %28, %29
  %31 = or i32 0, %27
  store i32 %30, i32* %6, align 4
  ; c = 0 xor v
  %32 = load i32, i32* %2, align 4
  %33 = xor i32 0, -1
  %34 = and i32 %32, %33
  %35 = xor i32 %32, -1
  %36 = and i32 0, %35
  %37 = or i32 %34, %36
  %38 = xor i32 0, %32
  store i32 %37, i32* %7, align 4
  ret i32 0
```

```
55 }
```

Listing 5.6: Instruction Substitution IR

5.2.2. Bogus Control Flow

Diese Verschleierungstechnik ist nach der Vorlage von Collberg et al. [7] implementiert. Die generelle Funktion dieser Technik ist in Kapitel 3.2.4 beschrieben. Die implementierte Verschleierungstechnik kann mit Hilfe von Parametern in ihrer Anwendung gesteuert werden. Der Parameter `-mllvm -bcf` aktiviert die Verwendung von „Bogus Control Flow“. Der Parameter `-bcf_loop=n` gibt an wie oft „Bogus Control Flow“ auf eine Funktion angewendet werden soll. Der Standardwert für diese Option ist n=1. Ein weiterer Parameter mit den Namen `-bcf_prob=m` legt fest, mit welcher Wahrscheinlichkeit ein Basic Block verschleiert werden soll. Wird der Parameter nicht angegeben, wird ein Wert von 30 angenommen. Folgend wird ein einfaches Beispiel für eine Verschleierung nach dieser Technik gezeigt. In Listing 5.7 ist der originale Quellcode zu sehen. Listing 5.9 zeigt den, von LLVM erzeugten IR Zwischencode ohne „Bogus Control Flow“. Die Variante mit der hier beschriebenen Technik ist in Listing 5.10 dargestellt.

Test der Funktion

Für den Test der Funktion wurde ein erweitertes „Hello World“ Programm geschrieben. Erweitert deshalb, weil die Ausgabe nicht in der Funktion `main()` sondern in einer eigenen Funktion `print()` ausgegeben wird. Dadurch kann die Funktion der Verschleierungstechnik besser gezeigt werden. Listing 5.7 zeigt den Quellcode des erweiterten „Hello World“ Programms.

```
1 #include<stdio.h>
2 void print(){
3   printf("Hello World!\n");
4 }
5 int main(){
6   print();
7   return 0;
8 }
```

Listing 5.7: Testprogramm „Hello World“ mit Funktion

In Listing 5.8 sind die verwendeten Befehle zum Kompilieren aufgelistet.

```
1 #LLVM Applikation
```

```
clang-4.0 helloWorld.c -o helloWorld.llvm
#LLVM IR Zwischencode
clang-4.0 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll

#"Bogus Control Flow" verschleierte Applikation
clang-4.0 helloWorld.c -mllvm -bcf -o helloWorld.llvmObf
#"Bogus Control Flow" verschleierter IR Zwischencode
clang-4.0 helloWorld.c -emit-llvm -S -mllvm -bcf -o helloWorld.llvmObf.ll
```

Listing 5.8: Befehle zum Kompilieren ohne und mit „Bogus Control Flow“

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define void @print() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}

declare i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @print()
  ret i32 0
}

```

Listing 5.9: LLVM IR ohne Verschleierung

Beim Vergleich von Listing 5.9 und Listing 5.10 ist zu sehen, wie zusätzliche Kontrollstrukturen in den Code eingefügt sind. Der von der Verschleierungstechnik eingefügte Code ist kommentiert, um damit zu zeigen welche Befehle eingefügt sind. „Bogus Control Flow“ wird mit einer Warscheinlichkeit von 30 % eingefügt. In diesem Fall wird nur in der Funktion `print()` die Technik „Bogus Control Flow“ eingefügt. Zu erkennen sind dabei eine If-Verzweigung, die immer auf `<label>:9:` verzweigt, weil der Teil die Variable `y=0` immer kleiner 10 ist und damit die `or`-Anweisung immer true liefert. Im 2. Teil ist eine Schleife zu erkennen, die nur einmal durchlaufen wird und dann die Funktion `print()` returniert.

```
@.str = private unnamed_addr constant [16 x i8] c"Hello World 1!\0A\00", align 1
@x = common global i32 0
@y = common global i32 0
@x.1 = common global i32 0
@y.2 = common global i32 0

; Function Attrs: noinline nounwind uwtable
define void @print() #0 {
%1 = load i32, i32* @x
%2 = load i32, i32* @y
```

```
%3 = sub i32 %1, 1
%4 = mul i32 %1, %3
%5 = urem i32 %4, 2
%6 = icmp eq i32 %5, 0
%7 = icmp slt i32 %2, 10    ; 0 < 10
%8 = or i1 %6, %7           ;
br i1 %8, label %9, label %20 ; if (%8) {label %9} else {label %20}

; <label>:9:                                     ; preds = %0, %20
%10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]* @.str, i32 0, i32 0))
%11 = load i32, i32* @x
%12 = load i32, i32* @y
%13 = sub i32 %11, 1
%14 = mul i32 %11, %13
%15 = urem i32 %14, 2
%16 = icmp eq i32 %15, 0
%17 = icmp slt i32 %12, 10    ; 0 < 10
%18 = or i1 %16, %17          ; %16 or true
br i1 %18, label %19, label %20 ; if (%8) {label %19} else {label %20}

; <label>:19:                                    ; preds = %9
ret void

; <label>:20:                                    ; preds = %9, %0
%21 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([16 x i8], [16 x i8]* @.str, i32 0, i32 0))
br label %9
}

declare i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind uwtable
define i32 @main() #0 {
%1 = alloca i32, align 4
store i32 0, i32* %1, align 4
call void @print()
ret i32 0
}
```

Listing 5.10: Obfuscator-LLVM IR mit „Bogus Control Flow“

Für die Untersuchungen wird der Decompiler Ghidra in der Version 9.0 verwendet. Listing 5.11 zeigt die dekompilierte Version der in Listing 5.10 dargestellten `print()` Funktion. Auch hier sind die If-Anweisung und die Schleife deutlich zu erkennen.

```
void print(void){
  if ((x * (x + -1) & 1U) == 0 || y < 10) goto LAB_00400551;
  do {
    printf("Hello World 1!\n");
    LAB_00400551:
    printf("Hello World 1!\n");
  } while ((x * (x + -1) & 1U) != 0 && 9 < y);
  return;
}
```

Listing 5.11: Dekompilierte Funktion `print()` nach der Anwendung von „Bogus Control Flow“

5.2.3. Control Flow Flattening

Die allgemeine Funktion von „Control Flow Flattening“ ist im Kapitel 3.2.5 beschrieben. Im Rahmen der LLVM Implementierung kann man die Funktion über 3 Parameter steuern. `-mllvm -fla` aktiviert die Verschleierung bei der Kompilierung. Die zusätzliche Option `-split` aktiviert das Zerteilen der Basic Blocks, was zu einer stärkeren Verschleierung führt. Mittels `-split_num=k` wird festgelegt, dass „Control Flow Flattening“ k-mal auf einen Basic Block angewendet wird. Der Standardwert liegt hierfür bei 1.

Test der Funktion

Für den Test der Funktion von „Control Flow Flattening“ wird abermals das Programm „Hello World“ mit einer zusätzlichen Funktion verwendet. Listing 5.12 zeigt den Quellcode des Programms.

```
#include<stdio.h>
void print(){
  printf("Hello World!\n");
}
int main(){
  print();
  return 0;
}
```

Listing 5.12: Testprogramm „Hello World“ mit Funktion

In Listing 5.13 sieht man die verwendeten Befehle für die folgenden IR Zwischencode Ausgaben. Bei diesen Befehle fällt auf, dass der Zusatz `-mllvm -split` verwendet wird. Ohne dem Parameter ist es nicht möglich, bei dem kurzen Beispielprogramm eine Anwendung der Technik „Control Flow Flattening“ zu erreichen.

```
#LLVM Applikation
clang-7 helloWorld.c -o helloWorld.llvm
#LLVM IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll

#"‘Control Flow Flattening"’ verschleierte Applikation
clang-7 helloWorld.c -mllvm -fla -mllvm -split -o helloWorld.llvmObf
#"‘Control Flow Flattening"‘ verschleierter IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -mllvm -fla -mllvm -split -o helloWorld.llvmObf.
```

```
    11
```

Listing 5.13: Befehle zum Kompilieren ohne und mit „Control Flow Flattening“, zusätzlich wurde der Parameter `-mllvm -split` verwendet

In dem folgenden Listing 5.14 ist das Ergebnis ohne Verschleierung durch „Control Flow Flattening“ dargestellt.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind uwtable
define void @print() #0 {
%1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
ret void
}

declare i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind uwtable
define i32 @main() #0 {
%1 = alloca i32, align 4
store i32 0, i32* %1, align 4
call void @print()
ret i32 0
}
```

Listing 5.14: LLVM IR ohne Verschleierung

In Listing 5.15 ist die Anwendung von „Control Flow Flattening“ auf das Testprogramm „Hello World“ zu sehen. Die Funktion `main()` besteht aus 2 Befehlen, dem Aufruf der Funktion `print()` und dem Befehl `return 0`. Normalerweise wären diese beiden Befehle ein Basisblock. Durch die Anwendung des Parameters `-mllvm -split` werden diese in 2 Basisblöcke zerteilt und „Control Flow Flattening“ kann darauf angewendet werden. Das Resultat ist, wie bereits in Abschnitt 3.2.5 beschrieben, als zentrales Element eine Switch-Anweisung. Diese steuert den Ablauf und ist verantwortlich die Basisblöcke in der richtigen Reihenfolge auszuführen. In der Funktion `main()` besteht die Switch-Anweisung aus zwei Einträgen, dem Basisblock für den Funktionsaufruf von `print()` und `return 0`. Zuerst wird der Wert zum Vergleichen gesetzt, im Listing als Wert 1 markiert. Dann wird die Switch-Anweisung durchlaufen. Der Wert 1 führt zum ersten Basisblock, dem Funktionsaufruf. Nach Beendigung von Block 1 wird wieder ein neuer Wert für die Switch-Anweisung gesetzt. Im Listing 5.15 ist dieser als Wert 2 markiert. Beim abermaligen Durchlaufen der Switch-Anweisung wird nun der Basisblock 2 ausgeführt. Auf die Funktion `print()` wurde auch „Control Flow Flattening“ angewendet. In dieser Funktion ist allerdings nur ein Befehl vorhanden und daher hat die Switch-Anweisung auch nur einen Eintrag.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
; Function Attrs: noinline nounwind uwtable
define void @print() #0 {
    %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
    %2 = alloca i32
    store i32 -1606598909, i32* %2      ;Wert 1 setzen
```

```
7     br label %3
8   ; <label>:3:
9     %4 = load i32, i32* %2
10    switch i32 %4, label %5 [        ; Switch-Anweisung
11    i32 -1606598909, label %6        ; Sprung zu Block 1
12    ]
13  ; <label>:5:
14    br label %7
15  ; <label>:6:
16    ret void                 ; return
17  ; <label>:7:
18    br label %3                ; Sprung zurück zu Switch
19 }
20 declare i32 @printf(i8*, ...) #1
21 ; Function Attrs: noinline nounwind uwtable
22 define i32 @main() #0 {
23    %1 = alloca i32, align 4
24    %2 = alloca i32
25    store i32 -626157998, i32* %2     ; Wert 1 setzen
26    br label %3
27  ; <label>:3:
28    %4 = load i32, i32* %2
29    switch i32 %4, label %5 [        ; Switch-Anweisung
30    i32 -626157998, label %6         ; Sprung zu Block 1
31    i32 -1024475263, label %7        ; Sprung zu Block 2
32    ]
33  ; <label>:5:
34    br label %8
35  ; <label>:6:                            ; Block 1
36    store i32 0, i32* %1, align 4
37    call void @print()             ; Funktionsaufruf print()
38    store i32 -1024475263, i32* %2       ; Wert 2 setzen
39    br label %8
40  ; <label>:7:                                ; Block 2
41    ret i32 0                ; return 0
42  ; <label>:8:
43    br label %3                ; Sprung zurück zu Switch
44 }
```

Listing 5.15: Obfuscator-LLVM IR mit „Control Flow Flattening“

Auch bei dieser Technnik wird mittels Decompiler Ghidra das binäre Resultat untersucht. Auffällig ist dabei, dass durch den Decompiler die Switch-Anweisungen nicht rekonstruiert werden. Es wird vermutet, dass erst bei mehreren Basisblöcken die Switch-Anweisung als solches erkannt wird. Dass eine Verschleierung stattgefunden hat, ist zumindest in der Funktion `main()` erkennbar. Die Funktion `print()` ist mit einem Befehl vermutlich zu kurz und die angewendete Verschleierung wird beim Prozess der Dekompilierung durch Optimierung entfernt.

```
1
2 void print(void){
3   printf("Hello World!\n");
4   return;
5 }
6
7 undefined8 main(void){
```

```
8   int local_10;
9
10  local_10 = 0xae882062;
11  do {
12    while (local_10 == -0x5177df9e) {
13      local_10 = 0x295e72f0;
14    }
15  } while (local_10 != 0x295e72f0);
16  print();
17  return 0;
18 }
```

Listing 5.16: Dekompilat des Testprogramms „Hello World“ verschleiert mit „Control Flow Flattening“

5.2.4. Basic Block Split

Basisblöcke sind Teil von Funktionen und sind gekennzeichnet durch einen Einsprungpunkt sowie einen Ausgangspunkt und können aus mehreren Befehlen bestehen. Die Verschleierungstechniken von Obfuscator-LLVM werden, je nach verwendetem Parameter, ein oder auch mehrmals auf Basisblöcke angewendet. Sind die Basisblöcke in einem Programm sehr groß, führt das zu einer geringen Verschleierung. Um Basisblöcke in kleinere, künstliche Blöcke zu unterteilen kann der Parameter `-mllvm -split` zu den Verschleierungstechniken hinzugefügt werden. Mit einem weiteren Parameter `-mllvm -split_num=1` kann bestimmt werden, wie oft ein Basisblock geteilt werden soll. Wird der Parameter nicht angegeben wird ein Defaultwert von 1 angenommen.

Test der Funktion

Die Funktion dieses Parameters wird bereits in der Anwendung von „Control Flow Flattening“ in Abschnitt 5.2.3 gezeigt.

5.2.5. Code Tamper-Proofing und Procedures Merging

Diese beiden Techniken sind nur in der kommerziellen Version von LLVM Obfuscation enthalten. Beide orientieren sich an den Standardtechniken nach Collberg et al. [7]. Da keine der beiden Techniken in der Open Source Variante vorhanden ist, gibt es dazu auch keinen Syntax. Ein Test ist dadurch auch nicht möglich.

5.2.6. Syntax Zusammenfassung

Listing 6.30 ist eine Zusammenfassung aller CFLAGS/CXXFLAGS Attribute mit welchen die Verschleierungstechniken nach Obfuscator-LLVM gesteuert werden können. Die Attribute in den eckigen Klammern sind optional und die angegebenen Werte repräsentieren die Defaultwerte wenn diese Option nicht angegeben wird.

```
#BogusControlFlow
-mllvm -bcf [-mllvm -bcf_loop=1 | -mllvm -bcf_prob=30]
#Control Flow Flattening
-mllvm -fla
#BasicBlockSplitting
-mllvm -split [-mllvm -split_num=1]
#Instruction Substitution
-mllvm -sub [-mllvm -sub_loop=1]
```

Listing 5.17: CFLAGS / CXXFLAGS der Befehlsersetzung

5.2.7. Funktion Annotation

Annotation erlaubt es, gezielt Funktionen mit ausgewählten Obfuscation-Techniken zu verschleiern. Vor jeder Funktion, welche verschleiert werden soll, wird eine Annotation, wie in Listing 5.18 ersichtlich, eingefügt. Die Annotation wird auch im Wiki [42] vom Obfuscator-LLVM beschrieben. Für jede Funktion können eine oder mehrere Annotationen angegeben werden.

```
int foo() __attribute((__annotate__(("fla"))));
int foo() {
  return 0;
}
```

Listing 5.18: Obfuscator-LLVM Annotation Beispiel

Eine Annotation kann auch negativ verfasst werden, um bestimmte Verschleierungstechniken zu deaktivieren. Ein Beispiel wäre hierfür „nofla“, um das „Control Flow Flattening“ zu deaktivieren.

6. Projekt „Hikari“

Dieses Projekt [43] setzt auf das zuvor beschriebene Projekt „Obfuscation-LLVM“ [34] auf. Die Autoren von Hikari haben sich 3 Design-Richtlinien als Grundlage genommen:

- **Leichte Portierbarkeit**: Die entwickelten Module sollen leicht portierbar sein, das heißt dass diese auch bei einer neuen Version von LLVM genutzt werden können.
- **Fokus liegt am Compiler**: Es wird vermieden, externe Bibliotheken zu verwenden. Es sollen nur Funktionen von LLVM verwendet werden.
- **Möglichst abstrakt**: Die Module werden für das Middle-End, den Optimierer, geschrieben, wo der Zwischencode verschleiert werden kann. Es wird vermieden, Transformationen am Backend durchzuführen, da sich sonst eine Plattformabhängigkeit ergeben kann.

Das Projekt Hikari hat die bereits in Obfuscator-LLVM implementierten Module übernommen und Fehler korrigiert. In Tabelle 6.1 ist eine Gegenüberstellung der umgesetzten Techniken und Funktionen der beiden Projekte zusammengefasst. Die letzte verfügbare Release von Hikari setzt auf der aktuellen LLVM-Compiler-Suite der Version 7 auf und ist daher aktueller als jene bei Obfuscator-LLVM, welche auf LLVM 4 basiert. Die Parameter zum Aktivieren und Steuern der Funktionen sind bei den beiden Projekten nicht kompatibel. Lediglich die Funktion Annotation ist gleich und im Fall von Hikari um die neuen Methoden erweitert worden. Zusätzlich bietet dieses Projekt auch eine Unterstützung von Funktion Annotation bei Objective-C Code an. Alle Verscheierungstechniken von Hikari sind im Wiki des Projekts [44] beschrieben. Die vom Vorgängerprojekt übernommenen Obfuscation-Techniken „Instruction Substitution“, beschrieben in Abschnitt 5.2.1, und „Control Flow Flattening“ / „Split Basic Block“, beschrieben in Abschnitt 5.2.3 und 5.2.4, werden an dieser Stelle nicht nochmal beschrieben, da im Zuge der Integration in Hikari nur leichte Modifikationen vorgenommen bzw. Fehler ausgebessert sind. Der abweichende Syntax zum Obfuscator-LLVM ist in Kapitel 6.4 dargestellt.
Zu beachten ist, dass sich die Open Souce Lizenz von NCSA (bei LLVM [45] und Obfuscator-LLVM [46]) auf GNU Affero General Public License Version 3 (AGPLV3) [47] geändert hat.

Obfuscation Technik	LLVM Obfuscator	Hikari
Instruction Substition	✔	✔
Bougus Control Flow	✔	↑
Control FLow Flattening	✔	✔
Basic Block Split	✔	✔
Anti Class Dump	✗	✔
Function Wrapper	✗	✔
String Encryption	✗	✔
Indirect Branching	✗	✔
Function Call Obfuscation	✗	✔
Functions Annotations	✔	✔

✔... vorhanden, ↑ ... verbessert, ✗ ... nicht vorhanden

Tabelle 6.1.: Diese Übersicht zeigt einen Vergleich der in Obfuscator-LLVM und Hikari implementierten Verschleierungstechniken und -funktionen.

6.1. Installation

Hikari wird nach der Anleitung vom github Projekt [25] installiert. Dazu müssen anfangs einige Applikationen nachinstalliert werden:

- Git - eine Software zur Versionsverwaltung
- SWIG - Schnittstellen Compiler zum Verbinden von C und C++ Programmen
- Python - Interpreter
- CMake - zum Erzeugen von Makefiles
- GCC - Compiler
- Clang - LLVM Compiler Frontend

Diese Applikationen und weitere benötigte werden mit dem Befehl aus Listing 6.1 installiert.

```
1 apt-get install gcc git cmake ninja-build g++ python-dev swig libedit-dev libxml2
    libxml2-dev clang-7 clang-tools-7 python
```

Listing 6.1: Paketinstallation für Hikari

Danach kann mit den Befehlen aus Listing 6.2 die Build Umgebung von Hikari installiert werden. Beim Builden von Hikari LLVM wird der Quellcode von Branch „release_70“ verwendet und das Ergebnis ins aktuelle Verzeichnis abgelegt.

```
git clone -b release_70 https://github.com/HikariObfuscator/Hikari.git Hikari &&
    mkdir Build && cd Build && cmake -G "Ninja" -DCMAKE_BUILD_TYPE=MinSizeRel -
    DLLVM_APPEND_VC_REV=on ../Hikari && ninja && ninja install && git clone https://
    github.com/HikariObfuscator/Resources.git ~/Hikari
```

Listing 6.2: Installation Hikari

Sind alle Befehle erfolgreich ausgeführt, werden 3 Verzeichnisse angelegt. Zwei davon, „Hikari“ und „Build“, befinden sich im aktuellen Verzeichnis. Im Verzeichnis „Hikari“ befindet sich der Sourcecode. Hier im Speziellen sei der Sourcecode der Obfuscation-Module erwähnt (Listing 6.3).

```
./Hikari/lib/Transforms/Obfuscation/AntiClassDump.cpp
./Hikari/lib/Transforms/Obfuscation/BogusControlFlow.cpp
./Hikari/lib/Transforms/Obfuscation/Flattening.cpp
./Hikari/lib/Transforms/Obfuscation/FunctionWrapper.cpp
./Hikari/lib/Transforms/Obfuscation/IndirectBranch.cpp
./Hikari/lib/Transforms/Obfuscation/SplitBasicBlocks.cpp
./Hikari/lib/Transforms/Obfuscation/StringEncryption.cpp
./Hikari/lib/Transforms/Obfuscation/Substitution.cpp
```

Listing 6.3: Quellcode der Obfuscation-Module

Im Verzeichnis Build/bin befinden sich die ausführbaren Dateien der neu gebildeten Compilerumgebung. Listing 6.4 zeigt jene Dateien, welche im Weiteren für das Kompilieren der Debian Quelldateien notwendig sind.

```
./Build/bin/clang++
./Build/bin/clang
```

Listing 6.4: Quellcode der Obfuscation-Module

Das dritte Verzeichnis befindet sich im Home Verzeichnis des aktuell verwendeten Benutzernamens und lautet „Hikari“. Darin befindet sich die Datei „SymbolConfig.json“. Diese dient zur Steuerung der Obfuscation-Funktion „FunctionCallObfuscate“ und enthält die Übersetzungen für die Funktionsnamen.

6.2. Verbessertes „Bogus Control Flow“

Diese Funktion wird von LLVM Obfuscator übernommen und verbessert. Die Auswertung der Verzweigungen ist bei der Vorgängerversion immer gleich und führt immer mit einem „true“ zum gewollten Code. In der umgesetzten Version von Hikari wird dies dynamischer gestaltet und der vorgesehene Wahrheitswert der Verzweigung wird im Vorhinein zufällig gewählt. Der Standardwert für die Wahrscheinlichkeit mit der ein Basic Block verschleiert wird, hat sich von 30 auf 70 erhöht.

Test der Funktion

Für den Test der Verschleierungstechnik „Bogus Control Flow“ wird, wie auch bei Obfuscator-LLVM, ein „Hello World“ Testprogramm mit zusätzlicher Funktion verwendet. Dieses Testprogramm ist in Listing 6.5 dargestellt.

```
#include<stdio.h>
void print(){
  printf("Hello World!\n");
}
int main(){
  print();
  return 0;
}
```

Listing 6.5: Testprogramm „Hello World“ mit Funktion

Im folgenden Listing 6.6 sind die für den Test verwendeten Befehle und Parameter aufgeführt.

```
#LLVM Applikation
clang-7 helloWorld.c -o helloWorld.llvm
#LLVM IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll

#"Bogus Control Flow" verschleierte Applikation
clang-7 helloWorld.c -mllvm -enable-bcfobf -o helloWorld.hikari
#"Bogus Control Flow" verschleierter IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -mllvm -enable-bcfobf -o helloWorld.hikari.ll
```

Listing 6.6: Befehle zum Kompilieren ohne und mit „Bogus Control Flow“

In Listing 6.7 ist die Übersetzung mittels LLVM ohne Verschleierung zu sehen.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @print() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}

declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @print()
  ret i32 0
}
```

Listing 6.7: LLVM IR ohne Verschleierung

Listing 6.8 zeigt die verschleierte Variante des „Hello World“ Testprogramms. In diesem Fall wird die Funktion `main()` mittels „Bogus Control Flow“ verschleiert. Beim mehrmaligen Übersetzen mit Hikari „Bogus Control Flow“ ist zu sehen, dass der Wahrheitswert der hinzugefügten Kontrollstrukturen zufällig gewählt wird.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
@LHSGV = private global i32 1846526411
@RHSGV = private global i32 -1158438935
@LHSGV.1 = private global i32 -1198188382
@RHSGV.2 = private global i32 169015846

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @print() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}

declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = load i32, i32* @LHSGV
    %2 = load i32, i32* @RHSGV
    %3 = xor i32 %1, %2
    %4 = add i32 %3, -294858849
    %5 = sub i32 %4, 894329737
    %6 = add i32 %5, 1066064558
    %7 = icmp ule i32 %6, 830227194
    br i1 %7, label %18, label %8
  ; <label>:8:                                      ; preds = %8, %0, %18
    %9 = load i32, i32* @LHSGV.1
    %10 = load i32, i32* @RHSGV.2
    %11 = xor i32 %9, %10
    %12 = sub i32 %11, -878174850
    %13 = add i32 %12, -202767134
    %14 = add i32 %13, 1632589239
    %15 = icmp ugt i32 %14, 1153797747
    %16 = alloca i32, align 4
    store i32 0, i32* %16, align 4
    call void @print()
    br i1 %15, label %8, label %17
  ; <label>:17:                                     ; preds = %8
    ret i32 0
  ; <label>:18:                                     ; preds = %0
    %19 = alloca i32, align 4
    store i32 0, i32* %19, align 4
    call void @print()
    br label %8
}
```

Listing 6.8: Hikari IR mit „Bogus Control Flow“

Auch bei dieser Technnik wird mittels Decompiler Ghidra das binäre Resultat untersucht. Es ist zu beachten, dass der Code in Listung 6.8 und Listing 6.9 nicht äquivalent ist, da die Verschleierungstechniken durch ihre Diversität bei jeder Ausführung leicht anderen Code erzeugen. Daher können diese beiden Codeteile nicht 1:1 verglichen werden.

```
  undefined8 main(void){
```

```
2   undefined4 *puVar1;
3   undefined4 local_28 [4];
4   undefined auStack24 [12];
5   int local_c;
6
7   puVar1 = (undefined4 *)auStack24;
8   if (((LLHSGV.3 ^ LRHSGV.4) + 0xd77dbc7c & 0x88eb31fd) == 0x1008ecec) {
9     puVar1 = local_28;
10    local_28[0] = 0;
11    print();
12  }
13  do {
14    local_c = ((LLHSGV.5 | LRHSGV.6) & 0x2f9f0cc5 ^ 0xfe30ebff) + 0xf600be90;
15    puVar1[-4] = 0;
16    *(undefined8 *)(puVar1 + -6) = 0x401235;
17    print(*(undefined *)(puVar1 + -6));
18    puVar1 = puVar1 + -4;
19  } while (local_c == 0x189fd067);
20  return 0;
21 }
```

Listing 6.9: Dekompilierte Funktion `main()` nach der Anwendung von „Bogus Control Flow„

6.3. Zusätzlich umgesetzte Verschleierungstechniken

Wie bereits erwähnt werden Verschleiungstechniken von Obfuscator-LLVM übernommen, verbessert oder Fehler behoben. Die folgenden Methoden werden neu implementiert.

6.3.1. Anti Class Dump

Dieses Modul wurde entwickelt, um die Arbeit des Tools class-dump [48] von Steve Nygard bei der Analyse von Ojective-C Code zu erschweren. Dabei werden Spuren von Ojective-C Klassen im IR Zwischencode entfernt, damit das Tool class-dump diese nicht extrahieren kann. Die Implementierung wird in 2 Stufen geplant, wobei erst die erste Stufe umgesetzt wurde. In dieser wird nur die Sektion `+initialize` betrachtet und Methoden modifiziert.

Test der Funktion

Diese Verschleierungstechnik kann nur mit Objective-C Code getestet werden. Normaler C oder C++ Code wird nicht verändert. Da Objective-C Code nicht im Fokus liegt, sind an dieser Stelle keine Tests

durchgeführt worden.

6.3.2. Function Wrapper

Diese Verschleierungstechnik erzeugt Funktionsattrappen, welche die eigentliche Funktion umschließen. Durch den Parameter `-fw_times` wird bestimmt, wie oft eine Funktion mit solch einer Attrappe umschlossen wird. Ohne Angabe des Parameters wird ein Defaultwert von 2 angenommen. Die Option `-fw_prob` gibt an, mit welcher Wahrscheinlichkeit eine Funktion überhaupt behandelt wird. Hierbei gibt es einen Defaultwert von 30. Damit diese Verschleierungstechnik angewendet wird, ist es notwendig sie mit dem Parameter `-mllvm -enable-funcwra` zu aktivieren.

Test der Funktion

Zum Testen wird, wie auch schon bei Obfuscator-LLVM aus Kapitel 5, ein erweitertes „Hello World“ Programm genutzt. Dieses enthält abgesehen von der Funktion `main()` eine weitere Funktion mit dem Namen `print()`, die „Hello World“ über die Bibliotheksfunktion `printf()` ausgibt. Mit diesem Beispiel wird gezeigt, wie Funktionen „gewrapped“ werden. Mit den Standardeinstellungen werden 30 % der Funktionen mit jeweils 2 Dummy-Funktionen gewrapped. In Listing 6.10 ist das Beispielprogramm „Hello World“ dargestellt.

```
#include<stdio.h>
void print(){
  printf("Hello World!\n");
}
int main(){
  print();
  return 0;
}
```

Listing 6.10: Testprogramm „Hello World“ mit Funktion

In Listing 6.11 sind die Befehle zum Erzeugen von vier verschiedenen Outputs angegeben. Einerseits wird eine nicht verschleierte Version erstellt, die in der binären und der IR Zwischencode Form vorliegt. Andererseits wird die verschleierte Version erzeugt, wieder in der binären und Zwischencode Form.

```
#LLVM Applikation
clang-7 helloWorld.c -o helloWorld.llvm
#LLVM IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll

#"Function Wrapper" verschleierte Applikation
```

```
clang-7 helloWorld.c -mllvm -enable-funcwra -o helloWorld.hikari
#"Function Wrapper" verschleierter IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -mllvm -enable-funcwra -o helloWorld.hikari.ll
```

Listing 6.11: Befehle zum Kompilieren ohne und mit „Function Wrapper“

Listing 6.12 zeigt den Code, welcher durch LLVM erzeugt wird. Dabei sind die 2 Funktionen `main()` und `print()` zu erkennen. Die Funktion `print()` ruft wiederum die externe Funktion `printf()` auf und gibt somit den Text „Hello World“ aus.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @print() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}
declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @print()
  ret i32 0
}
```

Listing 6.12: LLVM IR

Listing 6.13 zeigt den Code, welcher durch Hikari und die aktivierte Funktion „Function Wrapper“ erzeugt wird. Hier ist zu sehen, dass aus der Funktion `main()` nach wie vor die Funktion `print()` aufgerufen wird. In dieser Funktion wird nicht die externe Funktion `printf()` aufgerufen sondern die Dummy-Funktion `HikariFunctionWrapper.1()`. Diese ruft wiederum `HikariFunctionWrapper()` auf, bis diese Funktion schlussendlich die externe Funktion `printf()` aufruft. Damit wird der Aufruf der externe Funktion `printf()` zweimal verschachtelt.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
@llvm.compiler.used = appending global [2 x i8*] [i8* bitcast (i32 (i8*)* @HikariFunctionWrapper to i8*), i8* bitcast (i32 (i8
    *)* @HikariFunctionWrapper.1 to i8*)], section "llvm.metadata"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @print() #0 {
  %1 = call i32 @HikariFunctionWrapper.1(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}
declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @print()
  ret i32 0
}
define internal i32 @HikariFunctionWrapper(i8*) #1 {
  %2 = call i32 (i8*, ...) @printf(i8* %0)
```

```
20   ret i32 %2
21 }
22 define internal i32 @HikariFunctionWrapper.1(i8*) #1 {
23   %2 = call i32 @HikariFunctionWrapper(i8* %0)
24   ret i32 %2
25 }
```

Listing 6.13: Function Wrapper IR

Die dekompilierte Ansicht kann aus Listing 6.14 entnommen werden. Die Dekompilierung ist mit dem Tool Ghidra durchgeführt worden.

```
1  void print(void){
2    HikariFunctionWrapper.1();
3    return;
4  }
5  void HikariFunctionWrapper.1(void){
6    HikariFunctionWrapper();
7    return;
8  }
9  void HikariFunctionWrapper(void){
10   printf("Hello World!\n");
11   return;
12 }
13 undefined8 main(void){
14   print();
15   return 0;
16 }
```

Listing 6.14: Dekompiliertes Testprogramm nach der Anwendung von Hikari „Function Wrapper“

Mögliche Erweiterungen: Um Diversität in den Maschinencode zu bekommen, wäre es möglich den Parameter, welcher angibt wie oft verschachtelt wird, nicht fix, sondern in einem bestimmten Intervall zufällig zu wählen. Wird der verschleierte Code deassembeliert bleiben die Funktionsnamen erhalten wie aus dem Ergebnis der Dekompilierung von Tool Ghidra in Listing 6.14 oder von IDA in der Abbildung 6.1 ersichtlich ist. Daher kann man beim Auftreten von Funktionsnamen wie „HikariFunction-Wrapper“ darauf schießen, dass der Code verschleiert ist.

6.3.3. String Encryption

Diese Funktion sucht in allen globalen Variablen und Funktionen nach Zeichenketten und ersetzt diese durch eine Folge von Bytes, welche die verschlüsselte Repräsentation der Zeichenfolgen darstellt. Dieses Modul verfügt nur über einen einzelnen Parameter `-mllvm -enable-strcry`, welcher es erlaubt die Funktion zu aktivieren.

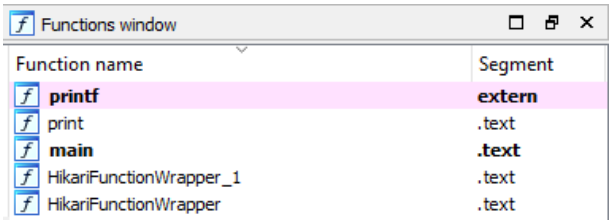

Abbildung 6.1.: von IDA Pro gefundene Funktionen im mit Function Wrapper verschleierten Binary

Test der Funktion

Für den Test kommt ein klassisches „Hello World“ Programm zum Einsatz. Dieses enthält eine Zeichenkette, die beim Verschleiern ersetzt wird. Listing 6.15 zeigt den Quellcode zum Testprogramm.

```c
#include<stdio.h>
int main(){
  printf("Hello World!\n");
  return 0;
}
```

Listing 6.15: Testprogramm „Hello World“

In Listing 6.16 sieht man die Befehle zum Erzeugen einer LLVM kompilierten Applikation, ihre zugehörigen Zwischencodes (IR) und das gleiche noch einmal für die verschleierte Version.

```
#LLVM Applikation
clang-7 helloWorld.c -o helloWorld.llvm
#LLVM IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll

#"String Encryption" verschleierte Applikation
clang-7 helloWorld.c -mllvm -enable-strcry -o helloWorld.hikari
#"String Encryption" verschleierter IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -mllvm -enable-strcry -o helloWorld.hikari.ll
```

Listing 6.16: Befehle zum Kompilieren ohne und mit „String Encryption“

Listing 6.17 zeigt die wesentlichen Teile des mit LLVM erzeugten IR Zwischencodes. Dabei ist zu erkennen, dass es eine globale Variable mit dem Namen `.str` gibt. Diese wird an die Funktion `printf()` übergeben.

```
@.str = private unnamed_addr constant [13 x i8] c"Hello World\0A\00", align 1
; Function Attrs: noinline nounwind optnone uwtable
```

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Listing 6.17: String Encryption LLVM IR

In Listing 6.18 ist das Ergebnis nach der Anwendung von „String Encryption“ zu sehen. Eine globale Variable `EncryptedString` wird angelegt. Diese enthält eine Bytefolge. In der Funktion `main()` wird, wenn die globale Variable „@0“den Wert 0 hat, das Label `<label>:3:` angesprungen. Dieser Block geht die Bytes einzeln durch und wendet ein `XOR` mit einem definierten Wert an. Das Ergebnis wird in die Bytefolge zurückgeschrieben. Ist der Block vollständig durchlaufen, steht in der globalen Variable `EncryptedString` die ursprüngliche Zeichenkette zur weiteren Verarbeitung zur Verfügung. Aus Gründen der Lesbarkeit wird [13 x i8], [13 x i8] durch [...] ersetzt.

```
@0 = private global i32 0
@EncryptedString = private global [13 x i8] c"\0D\1D\8Cz0\E2\FA>\C4ly\AC\B1"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = load atomic i32, i32* @0 acquire, align 4
  %2 = icmp eq i32 %1, 0
  br i1 %2, label %3, label %30

  ; <label>:3:                                      ; preds = %0
    %4 = load i8, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 0)
    %5 = xor i8 %4, 69
    store i8 %5, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 0)
    %6 = load i8, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 1)
    %7 = xor i8 %6, 120
    store i8 %7, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 1)
    %8 = load i8, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 2)
    %9 = xor i8 %8, -32
    store i8 %9, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 2)

    ... 27 ähnliche Zeilen gelöscht

    %28 = load i8, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 12)
    %29 = xor i8 %28, -79
    store i8 %29, i8* getelementptr inbounds ([...]* @EncryptedString, i32 0, i32 12)
    br label %30

  ; <label>:30:                                     ; preds = %0, %3
    store atomic i32 1, i32* @0 release, align 4
    %31 = alloca i32, align 4
    store i32 0, i32* %31, align 4
    %32 = getelementptr inbounds [...]* @EncryptedString, i32 0, i32 0
    %33 = call i32 (i8*, ...) @printf(i8* %32)
    ret i32 0
}
```

Listing 6.18: String Encryption Hikari IR

Mögliche Erweiterungen: Bei jedem Kompilierdurchgang werden die gleichen Werte für die `XOR` Verknüpfung verwendet. Würden die Werte zufällig gewählt, kann Diversität in den Maschinecode gebracht werden. Der neu gewählte globale Variablennamen `EncryptedString` für die verschlüsselte Zeichenkette gibt einen Hinweis, dass hier die Strings in irgendeiner Weise verschlüsselt wurden. Da sich die Methode nicht ändert und die Schlüsselwerte leicht extrahiert werden können, ist eine Rekonstruktion der Zeichenfolgen einfach möglich.

6.3.4. Indirect Branching

Um zu zeigen wie „Indirect Branching“ funktioniert wird ein zum klassischen „Hello World“ modifiziertes Testprogramm verwendet. In Listing 6.19 ist dieses Programm dargestellt. „Indirect Branching“ kann beim Kompilieren mittels des Parameters `-mllvm -enable-indibran` aktiviert werden. Bei dieser Verschleierungstechnik gibt es keine weiteren Parameter, um das Verhalten der Technik zu steuern. Listing6.20 zeigt die für den Test verwendeten Befehle.

```c
#include<stdio.h>
int main(int argc, char *argv[]){
  if (argc == 1){
    printf("Hello World!\n");
  } else {
    printf("Hello Earth!\n");
  }
  return 0;
}
```

Listing 6.19: Testprogramm „Hello World“

```
#LLVM Applikation
clang-7 helloWorld.c -o helloWorld.llvm
#LLVM IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll

#"‘Indirect Branching"’ verschleierte Applikation
clang-7 helloWorld.c -mllvm -enable-indibran -o helloWorld.hikari
#"‘Indirect Branching"’ verschleierter IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -mllvm -enable-indibran -o helloWorld.hikari.ll
```

Listing 6.20: Befehle zum Kompilieren ohne und mit „Indirect Branching“

Die unverschleierte Version des Testprogramms in der Ausprägung IR Zwischencode, ist in Listing 6.21 dargestellt. Es ist zu erkennen, dass jede if-Anweisung durch einen Vergleich mit folgendem Sprung (`br`) umgesetzt ist. Es ist leicht nachzuvollziehen, wann welcher Zweig ausgeführt wird.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
@.str.1 = private unnamed_addr constant [14 x i8] c"Hello Earth!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = load i32, i32* %4, align 4
  %7 = icmp eq i32 %6, 1          ; argc == 1
  br i1 %7, label %8, label %10   ; if(...)

  ; <label>:8:                                      ; preds = %2
    %9 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
    br label %12

  ; <label>:10:                                     ; preds = %2
    %11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str.1, i32 0, i32 0))
    br label %12

  ; <label>:12:                                     ; preds = %10, %8
    ret i32 0
}
```

Listing 6.21: LLVM IR ohne Verschleierung

Im Gegensatz dazu steht des Ergebnis der Verschleierung durch „Indirect Branching“. In Listing 6.22 ist das Ergebnis dargestellt. Wie bei der nicht verschleierten Version in Listing 6.21 kann auch hier der Vergleich `argc == 1` noch nachvollzogen werden. Während der nicht verschleierten Version ein Sprung folgt, wird der Sprung in der durch „Indirect Branching“ verschleierten Version mit einem indirekten Sprung durchgeführt. Die Sprungadresse kommt aus einer Sprungtabelle. Die Sprünge werden, bei Architekturen die das unterstützen, auf registerbasierte Sprünge umgewandelt.

```

@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
@.str.1 = private unnamed_addr constant [14 x i8] c"Hello Earth!\0A\00", align 1
@IndirectBranchingGlobalTable = internal global [3 x i8*] [i8* blockaddress(@main, %11), i8* blockaddress(@main, %14), i8* blockaddress(@main, %17)]
@HikariConditionalLocalIndirectBranchingTable = private global [2 x i8*] [i8* blockaddress(@main, %14), i8* blockaddress(@main, %11)]
@llvm.compiler.used = appending global [2 x i8*] [i8* bitcast ([3 x i8*]* @IndirectBranchingGlobalTable to i8*), i8* bitcast ([2 x i8*]* @HikariConditionalLocalIndirectBranchingTable to i8*)], section "llvm.metadata"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
```

```
  %6 = load i32, i32* %4, align 4
  %7 = icmp eq i32 %6, 1            ; argc == 1
  %8 = zext i1 %7 to i32
  %9 = getelementptr [2 x i8*], [2 x i8*]*  ; berechnet die Adresse in der Sprungtabelle
      @HikariConditionalLocalIndirectBranchingTable, i32 0, i32 %8
  %10 = load i8*, i8** %9
  indirectbr i8* %10, [label %14, label %11]

  ; <label>:11:                                     ; preds = %2
    %12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
    %13 = load i8*, i8** getelementptr inbounds ([3 x i8*], [3 x i8*]* @IndirectBranchingGlobalTable, i32 0, i32 2)
    indirectbr i8* %13, [label %17]

  ; <label>:14:                                     ; preds = %2
    %15 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str.1, i32 0, i32 0))
    %16 = load i8*, i8** getelementptr inbounds ([3 x i8*], [3 x i8*]* @IndirectBranchingGlobalTable, i32 0, i32 2)
    indirectbr i8* %16, [label %17]

  ; <label>:17:                                     ; preds = %14, %11
    ret i32 0
}
```

Listing 6.22: Hikari IR mit „Indirect Branching“

Wird die verschleierte Version des Testprogramms mittels Ghidra dekompiliert, zeigt sich folgendes Listing 6.23 . Der Decompiler hat versucht, die indirekten Sprünge als Switch-Anweisung darzustellen. Dabei kommt es wie in Listing 6.23 ersichtlich zu den Warnungen, dass die Springtabelle nicht rekonstruiert werden konnte. Die ursprüngliche Funktion des Programms wird jedoch korrekt wiederhergestellt. Wie das bei komplexeren Programmen aussieht, kann mit diesem einfachen Beispiel nicht beantwortet werden.

```
void main(int param_1,undefined8 param_2,undefined8 param_3){
  switch(param_1 == 1) {
    case false:
      printf("Hello Earth!\n",(&LHikariConditionalLocalIndirectBranchingTable)[(
    ulong)(param_1 == 1)],param_3,0);
        /* WARNING: Could not recover jumptable at 0x0040119b. Too many branches */
        /* WARNING: Treating indirect jump as call */
      (*IndirectBranchingGlobalTable._16_8_)(IndirectBranchingGlobalTable._16_8_);
      return;
    case true:
      printf("Hello World!\n",(&LHikariConditionalLocalIndirectBranchingTable)[(
    ulong)(param_1 == 1)],param_3,0);
        /* WARNING: Could not recover jumptable at 0x0040117d. Too many branches */
        /* WARNING: Treating indirect jump as call */
      (*IndirectBranchingGlobalTable._16_8_)(IndirectBranchingGlobalTable._16_8_);
      return;
  }
```

```
16 }
```

Listing 6.23: Mit „Indirect Branching“ verschleiertes Testprogramm dekompiliert

6.3.5. Function Call Obfuscation

Hikari bietet auch die Möglichkeit von „Function Call Obfuscation“. Die Funktion kann über den Parameter `-mllvm -enable-fco` aktiviert werden. Für diese Verschleierungstechnik wird eine Konfigurationsdatei im json Format benötigt. Hikari sucht diese Datei normalerweise unter /Hikari/SymbolConfig.json. Mit dem Parameter `-fcoconfig=PATH` kann eine Datei an einem anderen Ablageort angegeben werden. Mit dem Parameter `-fco_flag=VALUE` können die Werte `RTLD_GLOBAL|RTLD_NOW` auf der jeweilgen Plattform überschrieben werden. Die Datei SymbolConfig.json ist wie in Listing 6.25 dargestellt aufgebaut. Bei der Installation von Hikari wird eine Beispieldatei im Defaultverzeichnis abgelegt. Die in Listing 6.25 dargestellte Datei ist für das Testprogramm angepasst.

```
1 #include<stdio.h>
2 void print(){
3   printf("Hello World!\n");
4 }
5 int main(){
6   print();
7   return 0;
8 }
```

Listing 6.24: Testprogramm "Hello world“

```
1 {
2   "printf": "printf"
3 }
```

Listing 6.25: Konfigurationsdatei ~/Hikari/SymbolConfig.json

Um die verschleierte Version erfolgreich zu übersetzen, muss der Parameter `-ldl` angegeben werden. Die verwendeten Befehle für die Übersetzung des Testprogramms sind in Listing 6.26 dargestellt.

```
1 #LLVM Applikation
2 clang-7 helloWorld.c -o helloWorld.llvm
3 #LLVM IR Zwischencode
4 clang-7 helloWorld.c -emit-llvm -S -o helloWorld.llvm.ll
5
6 #"Function Call Obfuscation" verschleierte Applikation
```

```
clang-7 helloWorld.c -mllvm -ldl -enable-fco -o helloWorld.hikari
#"Function Call Obfuscation" verschleierter IR Zwischencode
clang-7 helloWorld.c -emit-llvm -S -ldl -mllvm -enable-fco -o helloWorld.hikari.ll
```

Listing 6.26: Befehle zum Kompilieren ohne und mit „Function Call Obfuscation“

Die Version ohne Verschleierung is bereits aus anderen Techniken bekannt. In Listing 6.27 ist der IR Zwischencode nochmals zum Vergleich mit der verschleierten Version angeführt.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @print() #0 {
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}

declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  call void @print()
  ret i32 0
}
```

Listing 6.27: LLVM IR Zwischencode ohne Verschleierung

In der mit „Function Call Obfuscation“ verschleierten Version werden die externen Funktionen mittles `dlopen` und `dlsym` verwendet. `dlopen` generiert einen Handler auf das Main-Programm und übergibt diesen an `dlsym`. Zusätzlich benötigt `dlsym` einen Zeichenkette mit dem Namen der externen Funktion. Dieser ist in der Variable @0 gespeichert. Mit diesen Informationen kann die Funktion `printf()` indirekt aufgerufen werden. In einem komplexeren Programm ist so nicht auf den ersten Blick ersichtlich, welche Funktionen aufgerufen werden.

```
@.str = private unnamed_addr constant [14 x i8] c"Hello World!\0A\00", align 1
@0 = private unnamed_addr constant [7 x i8] c"printf\00"  ; Name der externen Funktion

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @print() #0 {
%1 = call i8* @dlopen(i8* null, i32 258)
%2 = call i8* @dlsym(i8* %1, i8* getelementptr inbounds ([7 x i8], [7 x i8]* @0, i32 0, i32 0))
%3 = bitcast i8* %2 to i32 (i8*, ...)*
%4 = getelementptr inbounds [14 x i8], [14 x i8]* @.str, i32 0, i32 0
%5 = call i32 (i8*, ...) %3(i8* %4)             ; Indirekter Aufruf von printf()
ret void
}

declare dso_local i32 @printf(i8*, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
%1 = alloca i32, align 4
store i32 0, i32* %1, align 4
call void @print()
ret i32 0
```

```
22 }
23
24 declare i8* @dlopen(i8*, i32)
25
26 declare i8* @dlsym(i8*, i8*)
```

Listing 6.28: „Function Call Obfuscation“ Hikari IR Zwischencode

Bei Verwendung eines „State Of The Art“ Decompilers, wie es zum Beispiel das Programm Ghidra ist, kann der Funktionsaufruf gut ersichtlich zugeordnet werden. Es werden auch hier die Funktionen `dlopen` und `dlsym` verwendet, aber die Zeichenkette die die aufzurufende Funktion darstellt, wird im Klartext in der Funktion `dlsym` angegeben. In Listing 6.29 sieht man welche Funktion an dieser Stelle aufgerufen wird. Die Verschleierung ist hier nicht besonders effizient.

```
1  void print(void){
2    undefined8 uVar1;
3    code *pcVar2;
4
5    uVar1 = dlopen(0,0x102);
6    pcVar2 = (code *)dlsym(uVar1,"printf");
7    (*pcVar2)("Hello World!\n",pcVar2);
8    return;
9  }
10 undefined8 main(void){
11   print();
12   return 0;
13 }
```

Listing 6.29: Mit „Function Call Obfuscation“ verschleiertes Testprogramm dekompiliert

6.4. Syntax Zusammenfassung

Listing 6.30 ist eine Zusammenfassung aller CFLAGS/CXXFLAGS Attribute mit welchen die Verschleierungstechniken nach Hikari gesteuert werden können. Die Attribute in den eckigen Klammern sind optional und die angegebenen Werte repräsentieren die Defaultwerte wenn diese Option nicht angegeben wird.

```
1 #BogusControlFlow
2 -mllvm -enable-bcfobf [-mllvm -bcf_cond_compl=3 | -mllvm -bcf_prob=70 | -mllvm -
      bcf_loop=1]
3 #Control Flow Flattening
```

```
-mllvm -enable-cffobf
#BasicBlockSpliting
-mllvm -enable-splitobf [-mllvm -split_num=2]
#Instruction Substitution
-mllvm -enable-subobf [-mllvm -sub_loop=1 | -mllvm -sub_prob=50]
#AntiClassDump
-mllvm -enable-acdobf [-mllvm -acd-use-initialize]
#Register-Based Indirect Branching
-mllvm -enable-indibran
#String Encryption
-mllvm -enable-strcry
#Function Wrapper
-mllvm -enable-funcwra [-mllvm -fw_prob=30 | -mllvm -fw_times=2]
#Function Call Obfuscation
-mllvm -enable-fco [-fcoconfig=PATH | -fco_flag=VALUE]
#Function alle obfuskation Techniken
-enable-allobf
```

Listing 6.30: CFLAGS / CXXFLAGS der Befehlsersetzung

7. Herangehensweise

Für eine Untersuchung von verschleierten Programmen, sowohl statistisch als auch mit maschinellem Lernen, ist es notwendig, eine hohe Zahl an Beispielprogrammen zu analysieren. Um effizient maschinell zu lernen oder Algorithmen zu testen, ist es erforderlich, dass Beispielprogramme in der originalen und der verschleierten Form vorliegen. Ziel ist es, eine oder mehrere Quellen zu finden, welche es erlauben eine große Anzahl an Samples zu erzeugen. Diese sollen dabei ein breites Spektrum an Verschleierungstechniken abdecken. Dazu ist es notwendig, die geeigneten Quellen zu finden und eine Methode oder einen Compiler auszuwählen mit dem sich dieses Ziel erreichen lässt.

7.1. Auswahl von Quellprogrammen

Für die Kompilierung in der originalen und der verschleierten Form eines Programms wird der Quellcode benötigt. In der Open-Source-Software Gemeinde findet man eine große Anzahl von Applikationen für welche der Quellcode verfügbar ist. Um eine breite und repräsentative Auswahl von Applikationen zu bekommen, werden Programme des Open-Source Betriebssystems Linux benutzt. Diese stehen als C sowie C++ Quellcode und in ausreichender Anzahl zur Verfügung. Als Compiler ist die Wahl auf LLVM gefallen. Der Grund dafür ist in Abschnitt 7.2 erklärt. Debian Projekt „Rebuild of the Debian archive with clang“ beschäftigt sich mit der Recompilierung von Debian mit dem Compiler LLVM. Bei jeder neuen Version von Debian oder LLVM wird erneut versucht, die gesamte Disribution mit LLVM zu übersetzen. Demnach sollen 96 % aller Debian Pakete ohne Fehler mit LLVM/Clang in der Version 7 übersetzbar sein. Auf dieser Grundlage wird davon ausgegangen, dass eine Übersetzung der Quellpakete von Debian Linux einfach möglich ist.

7.2. Auswahl der Compilerumgebung und Verschleierung

Zum Verschleiern stehen die Obfuscation-Tools bezieheungsweise -Frameworks vom Abschnitt 2.7 zur Verfügung. Für die Verwendung in Forschungsarbeiten wird hier das Augenmerk auf frei verfügbare Software gelegt. Das ausgewählte Tool soll auch aktuell sein und wenn möglich aktiv entwickelt werden.

Die Wahl ist auf das Verschleierungstool „Hikari“ gefallen. In Kapitel 6 ist Hikari genauer beschrieben. Dieser Framework basiert auf dem Compiler LLVM, mit welchem Debian regelmäßig und zu einem hohen Prozentsatz erfolgreich übersetzt wird. Die aktuelle Version von Hikari basiert auf LLVM 7. In Debian 10 ist LLVM 7 das Standardpaket bei der Installation.

7.3. Sampleerzeugung

Die Sampleerzeugung erfolgt nach dem in Abbildung 7.1 dargestellten Prozess. Im ersten Schritt werden zu allen Dateien aus den Debian 10 Verzeichnissen /bin und /sbin die Quellpakete ermittel und eine Liste der zu kompilierenden Pakete erzeugt. Diese Liste ist der Input des Build-Scripts, welches den LLVM / Hikari Compiler steuert. Als weiterer Input fungieren die Quellpakte von Debian 10. Das Build-Script erzeugt aus den Quellpaketen mittels Obfuscation-Suite „Hikari“ binäre Dateien in unterschiedlichen Ausprägungen. Die Zuordnung der Dateiendungen zu den verwendeten Verschleierungstechniken ist in Tabelle 7.1 aufgelistet. Es sind folgende binäre Dateien für jedes Paket erzeugt worden:

- ohne Optimierung (-O0)
- mit Optimierung (-O2)
- ohne Optimierung (-O0) mit „Bogus Control Flow“
- ohne Optimierung (-O0) mit „Control Flow Flattening“
- ohne Optimierung (-O0) mit „Arithmentic Substitution“
- ohne Optimierung (-O0) mit „Indirect Branching“
- ohne Optimierung (-O0) mit „String Encryption“
- ohne Optimierung (-O0) mit „Function Wrapping“

Um den Effekt der Verschleierung nicht durch Optimierung des Codes zu verlieren, werden bei der Anwendung der verschiedenen Verschleierungstechniken die Optimierung deaktiviert. Zum Vergleich wird auch die binäre Variante ohne Verschleierung und Optimierung erzeugt. Die optimierte binäre Datei wird erstellt, da diese die Standardeinstellungen von den Compilern LLVM und gcc repräsentiert. Alle Verschleierungstechniken werden mit ihren Defaultparametern ausgeführt, da angenommen wird, dass die Obfuscation in den meisten Fällen so angewendet wird. Bei „Control Flow Flattening“ ist nicht zusätzlich die Basic Block Split angewendet, dies ist noch in weiteren Tests zu untersuchen. Die Technik „Anti Class Dump“ ist nur auf Objective-C Quelldateien anwendbar. Da es bei Linux in der Regel nur C und C++ Quelldateien gibt, ist diese Verschleierung nicht zur Anwendung gekommen.
Debian Quellpakete werden in der Regel mit make oder cmake übersetzt. Nicht alle Debian Quellpakete sind binär oder lassen sich mit make oder cmake übersetzen. Daher können einige Pakete für die Sampleerzeugung nicht genutzt werden. Das Ergebnis nach der Übersetzung wird anschließend noch in

Dateiendung	Verschleierungstechnik
llvmNoOpt	ohne Optimierung
llvmOpt	mit Optimierung
bcfobf	„Bogus Control Flow“
cffobf	„Control Flow Flattening“
funcwra	„Function Wrapping“
indibran	„Indirect Branching“
strcry	„String Encryption“
subobf	„Arithmentic Substitution“

Tabelle 7.1.: Zuordnung der Dateiendungen der erzeugten Binärdateien zu den Verschleierungstechniken

Bibliotheken und reguläre Programme sortiert und einer Analyse unterzogen, um festzustellen, ob die Verschleierung ordnungsgemäß angewendet ist.

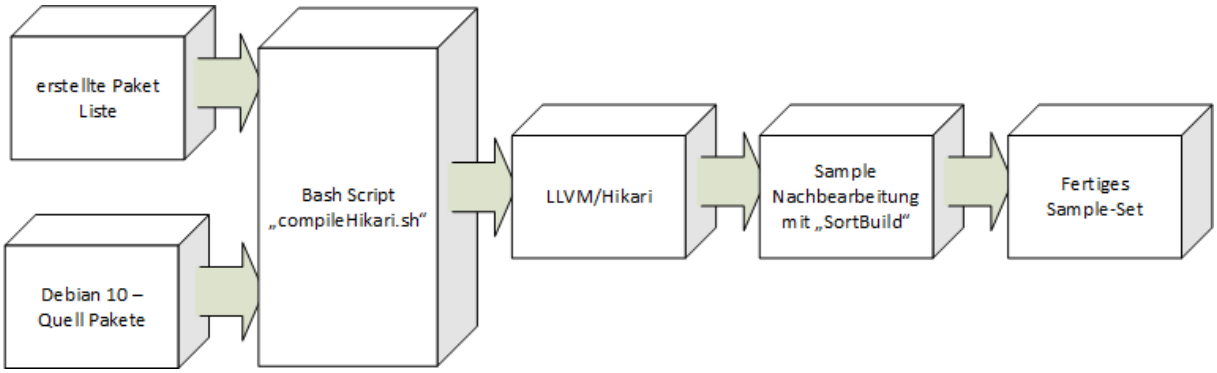

Abbildung 7.1.: Prozess der Sampleerzeugung

7.4. Verwendetes Build-System

7.4.1. Betriebssystem

Hikari basiert in der neuesten Version auf LLVM 7. Um eine einfachere Installation zu gewährleisten, wird eine Version vom Betriebssystem gewählt, welches out-of-the-box gleich die benötigte Version von LLVM unterstützt. Dabei ist die Wahl auf Debian 10 gefallen, welches zu diesem Zeitpunkt als „testing“ Release zur Verfügung steht. Debian 10 wird vom Image „netinst“ in der Minimal-Variante installiert. Alle benötigten Pakete werden bei Bedarf nachinstalliert.

7.4.2. Compiler - LLVM 7

Als Compiler wird LLVM in der Version 7 gewählt. Die Kernpakete von LLVM und das Frontend „Clang“ werden mit den Befehlen aus Listing 7.1 installiert.

```
apt-get install libllvm-7-ocaml-dev libllvm7 llvm-7 llvm-7-dev llvm-7-runtime
apt-get install clang-7 clang-tools-7
```

Listing 7.1: Paketinstallation für LLVM 7

7.4.3. Hikari

Die Installation von „Hikari“ ist im Abschnitt 6.1 beschrieben.

7.5. Erzeugung der Quellpaketliste

Die Textdatei, welche die Liste der zu übersetzenden Quellpakete enthält, ist manuell erstellt. Dazu werden die Befehle aus Listing 7.2 verwendet. Die ersten vier Befehle suchen zu jeder Datei in den Verzeichnissen /bin und /sbin den Paketnamen. Dieser wird an die Datei packages.txt angehängt. Da Pakete doppelt vorkommen können, wird die Liste mit dem fünften Befehl sortiert und Duplikate entfernt. Die daraus resultierende Liste wird als Input für das Build Script verwendet.

```
# Pakete der Programme aus /bin
dpkg -S /bin/* 2>&1|grep -v 'dpkg-query:'|cut -d':' -f1|sort|uniq>>packages.txt
# Pakete der Programme aus /sbin
dpkg -S /sbin/* 2>&1|grep -v 'dpkg-query:'|cut -d':' -f1|sort|uniq>>packages.txt
# Entfernend der doppelten Pakete
cat packages.txt |sort |uniq > packages_u.txt
```

Listing 7.2: Befehle für die Erzeugung der Quellpaketliste für das Build Script

7.6. Build Script

Der Prozess zum Herunterladen und Kompilieren der Debian Quellpakete ist automatisiert. Dazu wird das Script mit dem Namen „compileHikari.sh“ verwendet. Der Quellcode des Scripts ist in Anhang B angeführt.

7.6.1. Funktion

Die Aufgabe des Scripts ist es, die Quellpakete der Linux Distribution Debian 10, welche zeilenweise in einer Textdatei angegebenen sind, herunterzuladen und in den verschiedenen Ausprägungen zu kompilie-

ren. Das Script enthält mehrere Parameter, welche vor dem Ausführen konfiguriert werden müssen. Die Beschreibung der Parameter ist im Abschnitt 7.6.2 zu finden. Ist das Build Script konfiguriert, werden folgende Aktionen beim Ausführen durchgeführt:

1. **Auswahl des nächsten Pakets:** Das nächste Paket wird aus der Textdatei mit der Liste der Pakete gelesen.

2. **Umschalten des Standardcompilers auf LLVM:** normalerweise ist gcc als Compiler in den make und cmake Steuerungsdateien eingetragen. Daher ist es notwendig, die symbolischen Links `gcc`, `cpp` und `g++` auf die LLVM-Compiler Frontend Datei „clang“ zu setzen. Das dazugehörige Script „switch2llvm.sh“ ist in Listing 7.3 abgebildet und ist für das Build Script notwendig.

3. **Vorbereiten des Arbeitsverzeichnisses:** Aufräumen des Arbeitsverzeichnisses und download des Quellpaketes zur weiteren Verarbeitung

4. **Setzen der CFLAGS:** Das Kompilieren der Quellpakete kann über die Parameter `CFLAGS`, `CCFLAGS` und `CXXFLAGS` gesteuert werden. An dieser Stelle kann ausgewählt werden, welche Verschleierung angewendet wird. Über die Flags kann auch der Optimierer ein- bzw. ausgeschaltet werden. Es sei darauf hingewiesen, dass nicht alle Pakete mit diesen Flags gesteuert werden können und daher nicht alle Pakete das gewünschte Resultat liefern. Daher muss anschließend eine Nachbearbeitung stattfinden.

5. **Übersetzen des Pakets:** das Paket wird nach den Vorgaben der `CFLAGS` kompiliert. Dazu wird das Debian eigene Tool `debuild` verwendet.

6. **Verschieben der Ergebnisse ins Results-Verzeichnis:** Ist das Übersetzen durch `debuild` abgeschlossen, werden alle binären Dateien vom Verzeichnis des Parameters `DIR_WORKING` nach `DIR_RESULTS` verschoben.

7. **Nächste Verschleierungstechnik für dieses Paket:** Der Parameter `DO_OBFUSCATION` gibt die nächste Verschleierungtechnik vor. Die Verarbeitung wird bei Punkt 3 fortgesetzt.

8. **Nächstes Paket:** Sind alle Obfuscation-Techniken übersetzt worden, wird das nächste Paket ausgewählt und bei Punkt 4 fortgesetzt.

9. **Umschalten des Standardcompilers auf gcc:** Sind alle Pakete aus der Textdatei übersetzt, wird der Standardcompiler wieder auf gcc zurückgesetzt. Der Vorgang ist aus Listing 7.4 zu entnehmen. Es sollte vorher geprüft werden, welcher der Standardcompiler ist und das Script dementsprechend abgeändert werden.

```
#!/bin/bash
echo "Replace gcc, g++ & cpp by clang"
cd /usr/bin
rm g++ gcc cpp
ln -s /work/Build/bin/clang++ g++
ln -s /work/Build/bin/clang gcc
ln -s /work/Build/bin/clang cpp
```

Listing 7.3: Script „switch2llvm.sh“ zur Umschaltung des Default Compilers von gcc auf LLVM

```
#!/bin/bash
echo "Replace gcc, g++ & cpp by gcc"
cd /usr/bin
rm g++ gcc cpp
ln -s /usr/bin/g++-8 g++
ln -s /usr/bin/gcc-8 gcc
ln -s /usr/bin/cpp-8 cpp
```

Listing 7.4: Script „switch2gcc.sh“ zum Herstellen des Default Compilers gcc

7.6.2. Parameter

Folgende Auflistung zeigt die möglichen Parameter im Build Script und erklärt dessen Funktion:

- **FILE_PACKAGELIST:** Gibt den Ort der Textdatei mit den zeilenweise aufgelisteten Debian Quellpaketen an.
- **STARTPACKAGE:** Soll nicht die gesamte Liste der Quellpakete abgearbeitet werden, kann mit den Parametern `STARTPACKAGE` das erste Paket aus der Liste in der Datei `FILE_PACKAGELIST` festgelegt werden.
- **ENDPACKAGE:** Äquivalent zum vorherigen Parameter kann mit diesem das letzte Paket der Bearbeitung festgelegt werden.
- **DO_COMPILE:** Wird ein Startpaket festgelegt, dann muss dieser Parameter auf „no“ gesetzt werden. Wird kein Startpaket angegeben ist dieser auf „yes“ zu setzen.
- **DIR_WORKING:** Legt das Arbeitsverzeichnis fest.
- **DIR_RESULTS:** Legt das Verzeichnis für die Ausgabe fest.

- **DO_OBFUSCATION:** Enthält eine Liste der Hikari Parameter zum Aktivieren der Obfuscation-Techniken. Mögliche Werte in der Liste sind in der Tabelle 7.1 zu finden

7.7. Nachverarbeitung

Es kommt vor, dass die Übersetzung der Quellpakete fehlschlägt oder die Verschleierung nicht erwartungskonform angewendet wird. Daher müssen die erzeugten binären Dateien nachverarbeitet werden. Dazu wird ein Java-Programm mit dem Namen „SortBuild“ verwendet. Der Quellcode für dieses Programm ist in Anhang C abgebildet. „SortBuild“ sucht die zugehörigen binären Dateien zusammen und vergleicht die Regulären mit den Verschleierten. Kann kein Unterschied festgestellt werden, dann werden diese verworfen. Als Resultat einsteht ein Verzeichnis mit binären Dateien in allen Ausprägungen wie in Tabelle 7.1 beschrieben.

7.7.1. Funktion

Das Java-Programm besteht aus der Hauptklasse „SortBuild“. Darin befindet sich die `main()` Funktion. Die Funktion kann grob wie folgt beschrieben werden:

1. **Erzeuge eine Liste aller nicht verschleierten und nicht optimierten Dateien:** Zuerst wird eine Liste aller Dateien aus dem Verzeichnis `workingDir` mit der Endung `*.llvmNoOpt` erstellt.

2. **Erzeugen eines `BuildFileSet`:** Die Datei aus der zuvor erstellten Liste wird an die Klasse `BuildFileSet` übergeben. Diese sucht die dazugehörigen Dateien anhand des Dateinamens und der Dateiendung. Im `BuildFileSet` wird aus diesen Dateien eine Liste der Klasse `BuildFile` angelegt. Beim Anlegen einer neuen Klasse `BuildFile` wird die zugehörige Datei nach folgenden Kriterien analysiert:

   - Berechnung eines MD5 - Hash
   - Berechnung eines TLSH - Hash [49]
   - Suchen nach Zeichenketten, welche in den „Hikari“ Verschleierungstechniken vorkommen. In der Technik „String Encryption“ werden bei erfolgreicher Anwendung Zeichenketten wie „LEncryptedString“ gefunden. Die Technik „Function Wrapper“ hinterlässt „HikariFunctionWrapper“ Zeichenketten in der binären Datei. Bei „Indirect Branching“ wird „HikariConditionalLocalIndirectBranchingTable“ gefunden und die Technik „Bogus Control Flow“ hinterlässt „LHSGV“.

3. **Analyse des `BuildFileSet`:** Das BuildFileSet wird dann anhand der zuvor berechneten und erhobenen Dateieigenschaften analysiert. Dazu werden folgende Analysen innerhalb des Dateisets durchgeführt:

   - Distanz der TLSH-Hashes von den llvmNoOpt Dateien zu den Verschleierte
   - Ermittlung wie viele verschiedene MD5 - Hashes in dem Dateiset existieren
   - Berechnung der Durchschnittsdateigröße und die Differenz der einzelne Dateien.
   - Ermittlung ob die erwarteten Zeichenketten in den verschleierten Dateien gefunden werden.

   Ein Dateiset wird als vollständig angesehen, wenn alle MD5 Hashes der Dateigruppe unterschiedlich sind und die erwarteten Zeichenketten in den verschleierten Dateien gefunden werden.

4. **Verschieben der Dateigruppen:** Ist ein Dateiset vollständig, werden alle zugehörigen Dateien in das Verzeichnis `FullSet` verschoben. Alle anderen Dateien werden in verschiedene Verzeichnisse aufgeteilt, je nachdem welche Eigenschaften die Gruppe erfüllt.

7.7.2. Parameter

Das Programm „SortBuild“ kann über 2 Parameter in dieser Klasse konfiguriert werden. Diese sind in der folgenden Liste beschrieben:

- **workingDir:** Gibt den Ort der binären Dateien an, die analysiert werden sollen.
- **move:** Legt fest, ob die analysierten und zusammengehörigen Dateien in die entsprechenden Unterverzeichnisse verschoben werden sollen.

7.7.3. LLVM Obfuscation und Optimierung

Nach der Beschreibung von LLVM 4 [50] und 7 [51] verfügt `clang` aktuell über 8 Optimierungsstufen und 2 Alias. Diese sind von 0-4 durchnummeriert und 5 werden durch unterschiedliche Buchstaben gekennzeichnet. Die folgende Auflistung zeigt alle Optimierungstufen, die in LLVM 4 und 7 vorhanden sind.

- **-O0**: keine Optimierung
- **-O1**: zwischen Stufe 0 und 2
- **-O2**: moderate Optimierung; die meisten Optimierungen werden benutzt
- **-O3**: wie Stufe 2 mit zusätzlichen Optimierungen, die länger zum Kompilieren dauern oder längeren Maschinencode erzeugen im Hinblick auf schnellere Ausführungszeit
- **-O4**: Alias für -O3

- **-Ofast**: wie Stufe 3 mit zusätzlichen aggressiven Optimierungen, die unter Umständen strikte Regeln der Programmiersprache nicht einhalten.
- **-Os**: wie Stufe 2 im Hinblick auf Reduzierung vom Maschinencode
- **-Oz**: wie Stufe s mit stärkerer Reduzierung
- **-Og**: wie Stufe 1 mit verschiedenen Optimierungen deaktiviert, um die Informationen zum Debuggen zu erhalten
- **-O**: Alias für -O2

Es geht aus der Dokumentation nicht hervor, welche Optimierungsstufe angewendet wird, wenn kein Parameter zur Optimierung angegeben wird. Die Versuche haben gezeigt, dass ein Weglassen des Parameters zu keiner Optimierung (vergleichbar mit -O0) führt.

Repräsentativ für alle Optimierungsstufen und Versionen ist in den Listings 7.5 und Listing 7.6 aufgeführt, welche Optimierungsmodule beim Projekt Hikari basierend auf der LLVM Version 7 bei keiner Optimierung (-O0) und bei -O2 (moderat) angewendet werden. Entgegen der Standard LLVM Version 7 ist hier die Aktivierung der Verschleierungsmodule mit dem Parameter `-obfus` zu sehen.

```
1 llvm-as < /dev/null | opt -O0 -disable-output -debug-pass=Arguments
2 Pass Arguments:  -tti -verify -ee-instrument
3 Pass Arguments:  -targetlibinfo -tti -assumption-cache-tracker -profile-summary-info
      -obfus -forceattrs -basiccg -always-inline -barrier -verify
4 Loading Symbol Configuration From:/root/Hikari/SymbolConfig.json
5 Doing Post-Run Cleanup
6 Hikari Out
```

Listing 7.5: Befehl und Ausgabe zum Anzeigen der aktivierten Optimierungsmodule für die Stufe 0. Die Zeilen beginnend mit "Pass Arguments:“ zeigen alle aktivierten Module

```
1 llvm-as < /dev/null | /work/Build/bin/opt -O2 -disable-output -debug-pass=Arguments
2 Pass Arguments:  -tti -tbaa -scoped-noalias -assumption-cache-tracker -targetlibinfo
      -verify -ee-instrument -simplifycfg -domtree -sroa -early-cse -lower-expect
3 Pass Arguments:  -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker
      -profile-summary-info -obfus -forceattrs -inferattrs -ipsccp -called-value-
     propagation -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -
     loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -
     simplifycfg -basiccg -globals-aa -prune-eh -inline -functionattrs -domtree -sroa
      -basicaa -aa -memoryssa -early-cse-memssa -speculative-execution -basicaa -aa -
     lazy-value-info -jump-threading -correlated-propagation -simplifycfg -domtree -
     basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
```

```
    instcombine -libcalls-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-
    prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -loops -
    lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg
     -reassociate -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa
     -aa -scalar-evolution -loop-rotate -licm -loop-unswitch -simplifycfg -domtree -
    basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
    instcombine -loop-simplify
-lcssa-verification -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -
    loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -
    lazy-block-freq -opt-remark-emitter -gvn -phi-values -basicaa -aa -memdep -
    memcpyopt -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-branch-prob -lazy
    -block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-threading -
    correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-
    simplify -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -
    postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -
    lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -
    basiccg -rpo-functionattrs -globalopt -globaldce -basiccg -globals-aa -float2int
     -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa -scalar-
    evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-
    remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -
    basicaa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -
    opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa -loop-
    accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-
    remark-emitter -instcombine -simplifycfg -domtree -loops -scalar-evolution -
    basicaa -aa -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-
    emitter -slp-vectorizer -opt-remark-emitter -instcombine -loop-simplify -lcssa-
    verification -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block
    -freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa
     -scalar-evolution -licm -alignment-from-assumptions -strip-dead-prototypes -
    globaldce -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -
    lcssa-verification -lcssa -basicaa -aa -scalar-evolution -branch-prob -block-
    freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
    instsimplify -div-rem-pairs -simplifycfg -verify
Pass Arguments:  -targetlibinfo -domtree -loops -branch-prob -block-freq
Pass Arguments:  -targetlibinfo -domtree -loops -branch-prob -block-freq
Loading Symbol Configuration From:/root/Hikari/SymbolConfig.json
Doing Post-Run Cleanup
Hikari Out
```

Listing 7.6: Befehl und Ausgabe zum Anzeigen der aktivierten Optimierungsmodule für die Stufe 2. Die Zeilen beginnend mit "Pass Arguments:“ zeigen alle aktivierten Module.

Eine Beschreibung zu den einzelnen Modulen gibt es in der Dokumentation [52] von LLVM 7. Folgend wird die Auswirkung bei Verwendung von keiner (-O0) bzw. moderater (-O2) Optimierung gezeigt. Das Projekt „Hikari“ weist in der aktuellen Version des Wikis darauf hin, dass eine eingeschaltete Optimierung die Obfuscation zunichte machen kann. Um dies zu untersuchen, wird das Testprogramm aus Listing 7.7 verwendet. In diesem Testprogramm wird eine Addition berechnet und ausgegeben. Dieses wird einmal mit (-O2) und einmal ohne (-O0) Optimierung kompiliert. Anschließend werden die Unterschiede im IR Zwischenkode gezeigt. Die Befehle aus Listing 7.8 werden für die Kompilierung verwendet. Der Parameter `-mllvm -aesSeed=DEADBEEFDEADBEEFCAFEBABECAFEBABE` erlaubt, immer den gleichen SEED für den Pseudo-Zufallsgenerator zu verwenden. Dadurch wird keine Diversität bei diesem Test erzeugt und jedes Resultat ist im Bezug auf den Zufallsgenerator gleich.

```
#include<stdio.h>
int main(int argc, char* argv[]){
  int v = argc;
  int i;
  i = 10 + v;
  printf("add: %d", i);
  return 0;
}
```

Listing 7.7: Beispielprogramm für die Untersuchung der Auswirkung von Optimierung auf Code Verschleierung

```
#"Intruction Substitiution" verscheierte Applikation ohne Optimierung
clang-7 arithmetic.c -O0 -mllvm -enable-subobf -mllvm -aesSeed=DEADBEEFDEADBEEFCAFEBABECAFEBABE -o arithmetic.noopt
#"Intruction Substitiution" verscheierter IR Zwischencode ohne Optimierung
clang-7 arithmetic.c -O0 -mllvm -enable-subobf -mllvm -aesSeed=DEADBEEFDEADBEEFCAFEBABECAFEBABE -emit-llvm -S -o arithmetic.
    noopt.ll

#"Intruction Substitiution" verscheierte Applikation mit Optimierung
clang-7 arithmetic.c -O2 -mllvm -enable-subobf -mllvm -aesSeed=DEADBEEFDEADBEEFCAFEBABECAFEBABE -o arithmetic.opt
#"Intruction Substitiution" verscheierter IR Zwischencode mit Optimierung
clang-7 arithmetic.c -O2 -mllvm -enable-subobf -mllvm -aesSeed=DEADBEEFDEADBEEFCAFEBABECAFEBABE -emit-llvm -S -o arithmetic.
    opt.ll
```

Listing 7.8: Befehle zum Kompilieren ohne und mit Optimierung bei Anwendung von „Instruction Substitution“

Listing 7.9 zeigt ein typisches Ergebnis der Verschleierung einer Addition. Dabei werden die aus Tabelle 5.2 gezeigten Substitutionen angewendet.

```
@.str = private unnamed_addr constant [8 x i8] c"add: %d\00", align 1
```

```
2
3  ; Function Attrs: noinline nounwind optnone uwtable
4  define dso_local i32 @main(i32, i8**) #0 {
5  %3 = alloca i32, align 4
6  %4 = alloca i32, align 4
7  %5 = alloca i8**, align 8
8  %6 = alloca i32, align 4
9  %7 = alloca i32, align 4
10 store i32 0, i32* %3, align 4
11 store i32 %0, i32* %4, align 4    # argc
12 store i8** %1, i8*** %5, align 8  # argv
13 %8 = load i32, i32* %4, align 4
14 store i32 %8, i32* %6, align 4    # argc
15 %9 = load i32, i32* %6, align 4   # argc
16 %10 = sub i32 0, %9          # 0 - argc = - argc
17 %11 = sub i32 10, %10        # 10 - -argc
18 %12 = add nsw i32 10, %9      # 10 + argc
19 store i32 %11, i32* %7, align 4   # 10 - -argv
20 %13 = load i32, i32* %7, align 4  # 10 - -argv
21 %14 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]* @.str, i32 0, i32 0), i32 %13)
22 ret i32 0
23 }
```

Listing 7.9: Hikari IR Zwischencode mit Verschleierung „Instruction Substitution“ ohne Optimierung (-O0)

In Listing 7.10 ist die optimierte Form des Codes dargestellt. Dabei ist zu erkennen, dass der Optimierer nach der Verschleierung angewendet wird und den verschleierten Code wieder optimiert. In dem Listing ist nur noch die Addition und die Ausgabe zu sehen. Alle anderen, für die Verschleierung benötigten Instruktionen, werden wegoptimiert.

```
1  @.str = private unnamed_addr constant [8 x i8] c"add: %d\00", align 1
2
3  ; Function Attrs: nounwind uwtable
4  define dso_local i32 @main(i32, i8** nocapture readnone) local_unnamed_addr #0 {
5  %3 = add i32 %0, 10
6  %4 = tail call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]* @.str, i64 0, i64 0), i32 %3)
7  ret i32 0
8  }
```

Listing 7.10: Hikari IR Zwischencode mit Verschleierung „Instruction Substitution“ mit Optimierung (-O2)

Mit dieser Erkenntnis und dem Hinweis aus dem Wiki von Hikari, dass die Optimierung ausgeschaltet werden soll, wird die Optimierung bei der Übersetzung der Quellpakte ausgeschaltet.

8. Ergebnisse

In diesem Kapitel sind die Ergebnisse der Übersetzung der Debian Quellpakete zu finden.

8.1. Ermittelte Quellpakete

Die für die Übersetzung ausgewählten Quellpakete werden mit den Befehlen aus Listing 7.2 ermittelt. Die Verzeichnisse /bin und /sbin beinhalten jeweils 3123 und 342 Dateien. Aus diesen insgesamt 3465 Dateien werden durch die Befehle in Listing 7.2 481 Quellpakete für die Übersetzung mit LLVM und Hikari bestimmt. Von diesen Quellpaketen wurden 76 Pakete manuell entfernt. Es wird festgestellt, dass sich diese Pakete nicht erwartungsgemäß übersetzen lassen. Beispiele für Pakete, die entfernt werden sind: binutils, clang und cpp. Damit stehen 405 Pakete für die weitere Verarbeitung zur Verfügung.

8.2. Übersetzte binäre Dateien

Vom Build Script wird überprüft, ob sich ein Quellpaket mittels Konfigurationstools `configure` oder `cmake` konfigurieren lasst. Ist dies nicht der Fall, wird das Quellpaket ebenfalls ignoriert. Mit dieser Überprüfung werden weitere 153 Quellpakete ausgelassen. Damit werden 252 Quellpakete für die Erzeugung der binären Dateien verwendet. Die Übersetzung der verbleibenden Pakete dauert auf einem Quadcore Prozessor ca. 4 Tage.
Nach der Übersetzung werden alle binäre Dateien vom Build Script in ein Verzeichnis mit den Ergebnissen verschoben. Nach der vollständigen Übersetzung aller Quellpakete ergibt das 51625 Dateigruppen mit insgesamt 445697 Dateien. Diese binären Dateien beinhalten ausführbare Dateien und Bibliotheken. Im ersten Schritt werden diese aufgeteilt. Auf die Bibliotheken entfallen insgesamt 418240 Dateien. Die ausführbaren Dateien ergeben insgesamt 27457. Für die weitere Verarbeitung werden nur die ausführbaren Dateien betrachtet. Damit ergeben sich insgesamt 27457 ausführbare Dateien für die Nachverarbeitung.

8.3. Nachverarbeitung der ausführbaren Dateien

Die Nachverarbeitung wird mit dem in Abschnitt 7.7 beschriebenen Java-Tool „SortBuild“ durchgeführt. Als Input dienen alle, im vorherigen Schnitt erzeugten, 27457 ausführbaren Dateien. Nach dem erfolgreichen Anwenden des Tools „SortBuild“ sind die Dateien wie folgt aufgeteilt:

- Vollwertige Sets mit allen Verschleierungsmethoden angewendet:
  704 Dateigruppen oder 5632 Dateien insgesamt
- Volle Sets, ohne „String Encryption“:
  136 Dateigruppen oder 952 Dateien insgesamt
- Verschiedene Verschleierungsmethoden fehlgeschlagen
  348 Dateien insgesamt
- Nicht Verschleiert
  20525 Dateien insgesamt

Für eine Analyse bei der alle angewendeten Verschleierungsmethoden untersucht werden, können somit 704 Dateien mit 6 verschiedenen Obfuscation-Techniken verwendet werden. Dabei stehen Dateien mit einigen Kilobyte bis zu einigen zig Megabyte zur Verfügung.

8.4. Übersicht

In der Abbildung 8.1 ist die Übersicht der Anzahl der Dateien in den einzelnen Phasen der Erzeugung dargestellt.

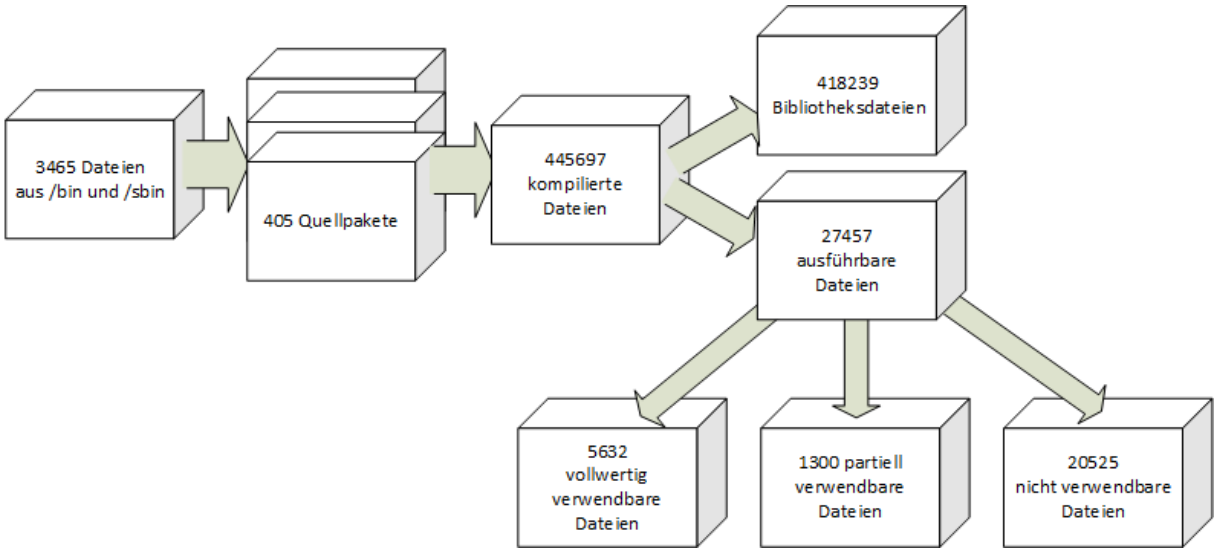

Abbildung 8.1.: Übersicht der erzeugten Dateien

9. Zusammenfassung und Ausblick

In diesem Kapitel werden die Erkenntnisse der Erzeugung von verschleierten Test-Samples zusammengefasst und ein Ausblick auf mögliche Erweiterungen gegeben.

9.1. Zusammenfassung

In dieser Arbeit wird die Erstellung von Testdaten zum Analysieren, Trainieren und Auswerten von Software zur Erkennung von Obfuscation behandelt. Dies beinhaltet die Auswahl eines geeigneten Tools zur Verschleierung, die Selektion der Eingangsdaten für die Verschleierung und die Aufbereitung der Testdaten für die anschließende Anwendung.
Zuerst wird ein Tool zum Verschlüsseln ausgewählt. Dabei ist die Wahl auf „Hikari“, eine LLVM basierende Compiler-Suite, gefallen. Diese ist in einer freien und in einer kommerziellen Version verfügbar. Hier kommt die freie Version zum Einsatz. Hikari ist ein aktives Projekt und wird auf die aktuellen Versionen von LLVM portiert.
Bei der Auswahl der Eingangsdaten für die Obfuscation-Suite werden die Quellpakete der Linux Distribution Debian verwendet. Die Auswahl der Pakete erfolgt nach den aktuell installierten Binärdateien auf der verwendeten Distribution. Die ausgewählten Quellpakete werden von einem entwickelten Buildscript mittels „Hikari“ kompiliert. Dabei wird die Optimierung bei der Übersetzung deaktiviert, da die, im Normalfall gewünschte, Optimierung eine negative Auswirkung auf die Verschleierung hat.
Nicht alle Quellpakete lassen sich erwartungsgemäß übersetzen. Daher müssen die erzeugten Testdaten vor der Verwendung nachverarbeitet werden. Bei dieser Verarbeitung werden Dateien, bei denen die Verschleierungstechniken nicht mit dem gewünschten Effekt angewendet worden sind, aussortiert. Als Ergebnis steht eine große Anzahl von Sets aus binären Dateien zur Verfügung. Diese Sets bestehen jeweils aus einer originalen, optimierten und nicht optimierten Datei und sechs verschleierten Versionen der Datei. Die angewendeten Verschleierungstechniken entsprechen den frei verfügbaren der Obfuscation-Suite „Hikari“.

9.2. Ausblick

Zum Zeitpunkt der Entstehung der Arbeit wird eine neue Version vom Verschleierungsframework Hikari released. Dabei handelt es sich um die Version release_80 vom 27. März 2019. Die wesentliche Änderung ist, dass Hikari nun auf der Compiler-Suite LLVM 8 basiert. Zusätzlich wird ein neuer Pseudo-Zufallsgenerator (PRNG) implementiert. Dieser liefert in bestimmten Fällen eine bessere Stabilität und Geschwindigkeit.

Im Rahmen der Arbeit sind nicht alle Verschleierungstechniken, welche Hikari bereitstellt, angewendet worden. Da in dieser Arbeit nur C und C++ Code zur Anwendung gekommen ist, kann in zukünftigen Untersuchungen die Technik „AntiClassDebug“ auf Objective-C Code angewendet werden.

Bei allen angewendeten Verschleierungstechniken der Obfuscation-Suite Hikari wurde der Defaultwert für die Parameter verwendet. Eine weitere Möglichkeit ist, die Techniken mit der Variation der Parameter zu optimieren.

Die beiden Techniken „Control Flow Flattening“ und „Split Basic Block“ können kombiniert angewendet werden um eine bessere Verschleierung zu erhalten, da die herkömmlichen Basisblöcke dadurch nochmals unterteilt werden und dadurch „Control Flow Flattening“ angewendet wird.

Um eine effizientere Erzeugung von Samples zu erzielen, ist es ratsam, die Quellpakete vorzuselektieren und eine Liste von Quellpaketen anzulegen, welche sich mit Verschleierung übersetzen lassen.

A. Abkürzungsverzeichnis

Abkürzung	Beschreibung
LLVM	früher Abkürzung für „Low Level Virtual Machine“
IR	Intermediate Representation
PRNG	Pseudo-Zufallsgenerator
AES128	Advanced Encryption Standard 128 bit
NIST	National Institute of Standards and Technology
PRNG	Pseudo-Zufallsgenerator

Tabelle A.1.: Abkürzungsverzeichnis

B. BuildScript

```
#!/bin/bash
LOGFILE="compileHikari.log"
FILE_PACKAGELIST="packages_u.txt"
STARTPACKAGE="kbd"
ENDPACKAGE=""
DO_COMPILE="no"
DIR_WORKING="/work/source/linuxBins/work1/"
DIR_RESULTS="/work/results/bin"
DO_OBFUSCATION="llvmOpt llvmNoOpt bcfobf cffobf splitobf subobf acdobf indibran
    strcry funcwra"
exec > $LOGFILE
exec 2>&1
while read CURRENTPACKAGE REST; do
  if [ "x$CURRENTPACKAGE" == "x$STARTPACKAGE" ]; then
    DO_COMPILE="yes"
  fi
  if [ "x$DO_COMPILE" == "xyes" ]; then
    echo "************ PACKAGE: $CURRENTPACKAGE"
    mkdir -p $DIR_WORKING
    # set llvm
    echo -e "Switch to llvm compiler... \c"
    /work/scripts/switch2llvm.sh && echo "OK" || echo "Not OK"
    for OBFUS in $DO_OBFUSCATION; do
      echo "*** compile $OBFUS"
      cd $DIR_WORKING
      rm -rf *
      echo -e "Get package... \c"
      apt-get -y source $CURRENTPACKAGE >/dev/null 2>&1 && echo "OK" || echo "Not OK
    "
      echo -e "Get dependencies... \c"
      apt-get -y build-dep $CURRENTPACKAGE >/dev/null 2>&1 && echo "OK" || echo "Not
     OK"
      #change into package directory
```

```
31      cd $(find . -maxdepth 1 -type d |grep /)
32      # Build with configure
33      if [ -f ./configure ] || [ -f ./CMakeLists.txt ]; then
34        echo "Setup configure"
35        if [ "x$OBFUS" == "xllvmNoOpt" ]; then
36          unset CFLAGS
37          unset CXXFLAGS
38          unset CCFLAGS
39          unset DEB_CFLAGS_MAINT_APPEND
40          unset DEB_CXXFLAGS_MAINT_APPEND
41          export DEB_BUILD_OPTIONS="nocheck noopt nodocs parallel=5"
42        elif [ "x$OBFUS" == "xllvmOpt" ]; then
43          unset CFLAGS
44          unset CXXFLAGS
45          unset CCFLAGS
46          unset DEB_CFLAGS_MAINT_APPEND
47          unset DEB_CXXFLAGS_MAINT_APPEND
48          export DEB_BUILD_OPTIONS="nocheck nodocs parallel=5"
49        else
50          export CFLAGS="-mllvm -enable-$OBFUS"
51          export CCFLAGS="-mllvm -enable-$OBFUS"
52          export CXXFLAGS="-mllvm -enable-$OBFUS"
53          export DEB_CFLAGS_MAINT_APPEND="-mllvm -enable-$OBFUS"
54          export DEB_CXXFLAGS_MAINT_APPEND="-mllvm -enable-$OBFUS"
55          export DEB_BUILD_OPTIONS="nocheck noopt nodocs parallel=5"
56        fi
57        echo -e "Build package... \c"
58        debuild -b -us -uc >/dev/null 2>&1 && echo "OK" || echo "Not OK"
59        echo -e "Copy bins... \c"
60        COUNTER=0
61        for FILE_BIN in $(find ./ -type f -exec file {} \;|grep ELF|cut -d : -f1);
    do
62          cp $FILE_BIN "$DIR_RESULTS/$(basename $FILE_BIN).$OBFUS"
63          COUNTER=$(($COUNTER+1))
64        done
65        echo "$COUNTER Files copied"
66      else
67        echo "$CURRENTPACKAGE: skipped"
68      fi
69    done
70    # reset gcc
```

```
71      /work/scripts/switch2gcc.sh
72      cd $DIR_WORKING
73      rm -rf *
74    fi
75    if [ "x$CURRENTPACKAGE" == "x$ENDPACKAGE" ]; then
76      DO_COMPILE="no"
77      exit 0
78    fi
79  done < $FILE_PACKAGELIST
```

Listing B.1: Buildscript für Hikari und LLVM

C. Nachbearbeitungsprogramm „SortBuild“

```java
package net.kraftl.fh.da.sortbuild;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.util.Collection;
import java.util.HashMap;
import org.apache.commons.io.FileUtils;
import org.apache.commons.io.filefilter.WildcardFileFilter;
import com.trendmicro.tlsh.Tlsh;
import com.trendmicro.tlsh.TlshCreator;

public class SortBuild {
  static String workingDir = "e:\\STORE\\DA\\work\\test\\test2\\";
  static int fullCount = 0;
  static boolean move = true;

  private static Collection<File> getAllNonOptimizedFileCollection(String folderName
    ) {
  File folder = new File(folderName);
  return FileUtils.listFiles(folder, new WildcardFileFilter("*.llvmNoOpt"), null);
  }

  public static void main(String[] args) {
    HashMap<Integer, Integer> countList = new HashMap<Integer, Integer>();
    Collection<File> nonOptimizedFiles = getAllNonOptimizedFileCollection(workingDir
    );
    for (File nonOptimizedFile : nonOptimizedFiles) {
      System.out.println("File: " + nonOptimizedFile.getName());
      BuildFileSet buildFileSet = new BuildFileSet(nonOptimizedFile);
      if (buildFileSet.getStatus() == SetStatus.FULL) {
```

```
32          fullCount++;
33          if (move) {
34            new File(workingDir + "\\FullSet").mkdirs();
35            buildFileSet.moveFileSet(workingDir + "\\FullSet");
36          }
37        } else {
38          if (move) {
39            // Alle gleich
40            if (buildFileSet.getDiffHash() == 1 && buildFileSet.getFileCount() == 7) {
41              new File(workingDir + "\\---AllSame").mkdirs();
42              buildFileSet.moveFileSet(workingDir + "\\---AllSame");
43            } else if (buildFileSet.getDiffHash() == 2 && buildFileSet.getFileCount()
    == 7) {
44              // NoOpt und alle Obf gleich
45              new File(workingDir + "\\---NoOptAndAllObfSame").mkdirs();
46              buildFileSet.moveFileSet(workingDir + "\\---NoOptAndAllObfSame");
47            } else if (buildFileSet.isNotObfus()) {
48              new File(workingDir + "\\---NoObfus").mkdirs();
49              buildFileSet.moveFileSet(workingDir + "\\---NoObfus");
50            } else {
51              new File(workingDir + "\\PartSet").mkdirs();
52              buildFileSet.moveFileSet(workingDir + "\\PartSet");
53            }
54          }
55          if (countList.containsKey(buildFileSet.getDiffHash())) {
56            countList.put(buildFileSet.getDiffHash(), countList.get(buildFileSet.
    getDiffHash()) + 1);
57          } else {
58            countList.put(buildFileSet.getDiffHash(), 1);
59          }
60        }
61      }
62      System.out.println("FullCount: " + fullCount);
63      for (HashMap.Entry<Integer, Integer> entry : countList.entrySet()) {
64        System.out.println("diffHahes: " + entry.getKey() + " times: " + entry.
    getValue());
65      }
66    }
67  }
```

Listing C.1: Buildscript für Hikari und LLVM

```
package net.kraftl.fh.da.sortbuild;

public enum SetStatus {
  FULL,
  NotFULL
}
```

Listing C.2: Buildscript für Hikari und LLVM

```
package net.kraftl.fh.da.sortbuild;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.ArrayList;
import java.util.HashSet;
import com.trendmicro.tlsh.Tlsh;

public class BuildFileSet {
  String[] extList = { "llvmNoOpt", "llvmOpt", "bcfobf", "cffobf", "funcwra", "
    indibran", "subobf" };
  ArrayList<BuildFile> fileList = new ArrayList<BuildFile>();
  SetStatus status;
  int fileCount = 0;
  int diffHash = 0;
  long fileSizeAvg = 0;
  boolean notObfus = false;
  public BuildFileSet(File fileNameWithExt) {
    for (String ext : extList) {
      fileList.add(new BuildFile(fileNameWithExt, ext));
    }
    analyse();
    for (BuildFile f : fileList) {
      f.getFileAttrString();
      if (f.getExtension().equalsIgnoreCase("strcry") || f.getExtension().
    equalsIgnoreCase("bcfobf") || f.getExtension().equalsIgnoreCase("indibran") || f
    .getExtension().equalsIgnoreCase("funcwra")) {
        if (f.getOccurrenceOfString() == 0) {
```

```
29            notObfus = true;
30          }
31        }
32      }
33      System.out.println("fileSizeAvg:" + fileSizeAvg);
34    }
35    private BuildFile getExtension(String ext) {
36      for (BuildFile f : fileList) {
37        if (f.getExtension().equalsIgnoreCase(ext)) {
38          return f;
39        }
40      }
41      return null;
42    }
43    private void analyse() {
44      int totalCount = extList.length;
45      // Check distance of TLSH Hashes
46      Tlsh nonOpt_tlsh = getExtension("llvmNoOpt").getTlsh_hash();
47      for (BuildFile f : fileList) {
48        if (!f.getExtension().equalsIgnoreCase("llvmNoOpt")) {
49          if (f.getTlsh_hash() != null) {
50            f.setLlvmNoOpt_hash_diff(nonOpt_tlsh.totalDiff(f.getTlsh_hash(), false));
51          }
52        }
53      }
54      // Build HashSet of all Hashes and then check the size of the HasSet
55      HashSet<String> h = new HashSet<String>();
56      for (BuildFile f : fileList) {
57        if (f.getHash() != null) {
58          h.add(f.getHash());
59          fileCount++;
60        }
61      }
62      diffHash = h.size();
63      // Check if all Hashes different
64      if (fileCount == totalCount && diffHash == fileCount) {
65        status = SetStatus.FULL;
66      } else {
67        status = SetStatus.NotFULL;
68      }
69      // Calc avg fileSize and write diff to files
```

```
70      long sum = 0;
71      int count = 0;
72      // average filesize
73      for (BuildFile f : fileList) {
74        sum += f.getSize();
75        count++;
76      }
77      fileSizeAvg = sum / count;
78      // Calc diff
79      for (BuildFile f : fileList) {
80        f.setFileSizeDiff2Avg(f.getSize() - fileSizeAvg);
81      }
82      for (BuildFile f : fileList) {
83        if (f.getOccurrenceOfString() == 0) {
84          status = SetStatus.NotFULL;
85        }
86      }
87    }
88    public SetStatus getStatus() {
89      return status;
90    }
91    public int getFileCount() {
92      return fileCount;
93    }
94    public int getDiffHash() {
95      return diffHash;
96    }
97    public void moveFileSet(String folder) {
98      try {
99        BuildFile bf = getExtension("llvmNoOpt");
100       for (BuildFile f : fileList) {
101         if (f.isExists()) {
102           String dup = "";
103           if (!f.getExtension().equalsIgnoreCase("llvmNoOpt") &&        bf.getHash()
    .equalsIgnoreCase(f.getHash())) {
104             dup = ".---";
105           }
106           Files.move(Paths.get(f.getFullFilename()),
107           Paths.get(folder + "\\" + f.getBaseName() + "." + f.getExtension() + dup),
108           StandardCopyOption.REPLACE_EXISTING);
109         }
```

```
110       }
111     } catch (IOException e) {
112       // TODO Auto-generated catch block
113       e.printStackTrace();
114     }
115   }
116   public boolean isFilePresent(String extention) {
117     for (BuildFile f : fileList) {
118       if (f.getExtension().equalsIgnoreCase(extention) && f.isExists()) {
119         return true;
120       }
121     }
122     return false;
123   }
124   public boolean isNotObfus() {
125     return notObfus;
126   }
127   public void setNotObfus(boolean notObfus) {
128     this.notObfus = notObfus;
129   }
130 }
```

Listing C.3: Buildscript für Hikari und LLVM

```
1  package net.kraftl.fh.da.sortbuild;
2
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.FileNotFoundException;
6  import java.io.IOException;
7  import java.io.InputStream;
8  import java.nio.file.Files;
9  import java.nio.file.Paths;
10 import java.util.Scanner;
11 import org.apache.commons.io.FilenameUtils;
12 import com.trendmicro.tlsh.Tlsh;
13 import com.trendmicro.tlsh.TlshCreator;
14
15 public class BuildFile {
16   boolean exists = false;
17   String hash = null;
```

```
18  Tlsh tlsh_hash = null;
19  int llvmNoOpt_hash_diff = -1;
20  String baseName;
21  String extension;
22  String folder;
23  long size = 0;
24  long fileSizeDiff2Avg = 0;
25  int occurrenceOfString = -1;
26  public BuildFile(File file, String extension) {
27    baseName = FilenameUtils.getBaseName(file.getName());
28    folder = FilenameUtils.getFullPath(file.getAbsolutePath());
29    this.extension = extension;
30    File f = new File(getFullFilename());
31    if (f.exists() && !f.isDirectory()) {
32      exists = true;
33      size = f.length();
34      try {
35        hash = calcHash(getFullFilename());
36        tlsh_hash = calculateTLSH(getFullFilename());
37      } catch (IOException e) {
38        hash = null;
39      }
40    }
41    // read file and find text
42    if (extension.equalsIgnoreCase("strcry") || extension.equalsIgnoreCase("bcfobf")
43    || extension.equalsIgnoreCase("indibran") || extension.equalsIgnoreCase("funcwra
      ")) {
44      occurrenceOfString = 0;
45      int counter = 0;
46      String searchString = "wiejiwejijweifjwiefji";
47      switch (extension) {
48        case "strcry":
49          searchString = "LEncryptedString";
50        break;
51        case "funcwra":
52          searchString = "HikariFunctionWrapper";
53        break;
54        case "indibran":
55          searchString = "HikariConditionalLocalIndirectBranchingTable";
56        break;
57        case "bcfobf":
```

```
58            searchString = "LHSGV";
59          break;
60        }
61        if (exists) {
62          try {
63            byte[] bytes = Files.readAllBytes(Paths.get(getFullFilename()));
64            String content = new String(bytes);
65            for (int index = content.indexOf(searchString, 0); index != -1; index =
    content
66                .indexOf(searchString, index + 1)) {
67              counter++;
68            }
69          } catch (IOException e) {
70            // TODO Auto-generated catch block
71            e.printStackTrace();
72          }
73        }
74        occurrenceOfString = counter;
75      }
76    }
77    public String getFullFilename() {
78      return folder + baseName + "." + extension;
79    }
80    private String calcHash(String file) throws IOException {
81      InputStream is = new FileInputStream(new File(file));
82      String hash = org.apache.commons.codec.digest.DigestUtils.md5Hex(is);
83      is.close();
84      return hash;
85    }
86    private static Tlsh calculateTLSH(String inputfile) throws IOException {
87      TlshCreator tlshCreator = new TlshCreator();
88      byte[] buf = new byte[1024];
89      InputStream is;
90      is = new FileInputStream(new File(inputfile));
91      int bytesRead = is.read(buf, 0, buf.length);
92      while (bytesRead >= 0) {
93        tlshCreator.update(buf, 0, bytesRead);
94        bytesRead = is.read(buf, 0, buf.length);
95      }
96      is.close();
97      Tlsh hash = tlshCreator.getHash();
```

```
98      return hash;
99    }
100   public void getFileAttrString() {
101     System.out.println("*** BaseName:\t" + baseName);
102     System.out.println("Extension:\t" + extension);
103     // System.out.println("**** Extension:\t" + extension);
104     // System.out.println("Folder:\t" + folder);
105     System.out.println("Exists:\t" + exists);
106     System.out.println("Size:\t" + size);
107     System.out.println("Hash:\t" + hash);
108     System.out.println("TLSH-Hash:\t" + tlsh_hash);
109     System.out.println("llvmNoOpt_hash_diff:\t" + llvmNoOpt_hash_diff);
110     System.out.println("fileSizeDiff2Avg:\t" + fileSizeDiff2Avg);
111     System.out.println("occurrenceOfString:\t" + occurrenceOfString);
112   }
113   public String getHash() {
114     return hash;
115   }
116   public String getBaseName() {
117     return baseName;
118   }
119   public String getExtension() {
120     return extension;
121   }
122   public String getFolder() {
123     return folder;
124   }
125   public long getSize() {
126     return size;
127   }
128   public boolean isExists() {
129     return exists;
130   }
131   public Tlsh getTlsh_hash() {
132     return tlsh_hash;
133   }
134   public int getLlvmNoOpt_hash_diff() {
135     return llvmNoOpt_hash_diff;
136   }
137   public void setLlvmNoOpt_hash_diff(int llvmNoOpt_hash_diff) {
138     this.llvmNoOpt_hash_diff = llvmNoOpt_hash_diff;
```

```
139   }
140   public long getFileSizeDiff2Avg() {
141     return fileSizeDiff2Avg;
142   }
143   public void setFileSizeDiff2Avg(long fileSizeDiff2Avg) {
144     this.fileSizeDiff2Avg = fileSizeDiff2Avg;
145   }
146   public int getOccurrenceOfString() {
147     return occurrenceOfString;
148   }
149   public void setOccurrenceOfString(int occurrenceOfString) {
150     this.occurrenceOfString = occurrenceOfString;
151   }
152
153 }
```

Listing C.4: Klasse BuildFile LLVM

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] A. Stepper, M. Höschel, and P. Wilhelm, "Ausarbeitung patentrecht," 2010.

[2] "Patentgesetz von 1970 Österreich," https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=10002181, accessed: 2019-05-05.

[3] "Richtlinien zur bearbeitung computerimplementierter erfindungen richtlinien zur bearbeitung von anmeldungen zu computerimplementierten erfindungen (software)," https://www.patentamt.at/fileadmin/root_oepa/Dateien/Patente/PA_Infoblaetter/PA_Richtlinien_Software_02082006.pdf, accessed: 2019-04-27.

[4] "Wikipedia softwarepatente," https://de.wikipedia.org/wiki/Softwarepatent, accessed: 2019-04-27.

[5] "List of most expensive video games to develop," https://en.wikipedia.org/wiki/List_of_most_expensive_video_games_to_develop, accessed: 2019-03-16.

[6] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[7] J. Nagra and C. Collberg, *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[8] J. A. Bloom, I. J. Cox, T. Kalker, J.-P. Linnartz, M. L. Miller, and C. B. S. Traw, "Copy protection for dvd video," *Proceedings of the IEEE*, vol. 87, no. 7, pp. 1267–1276, 1999.

[9] M. Becker and A. Desoky, "A study of the dvd content scrambling system (css) algorithm," in *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology, 2004.* IEEE, 2004, pp. 353–356.

[10] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 41–47, 2011.

[11] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 222–235. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274726

[12] R. R. Kiddy, "Method of obfuscating computer instruction streams," Feb. 17 2004, uS Patent 6,694,435.

[13] B. R. Deepas, B. Chandan Kumar, and B. D. Laitha, "Code obfuscation by using array transformation techniques," *IRACST - International Journal of Computer Science and Information Technology & Security (IJCSITS)*, vol. 7, no. 6, 2017.

[14] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 2012, pp. 9–16.

[15] M. Mateas and N. Montfort, "A box, darkly: Obfuscation, weird languages, and code aesthetics," in *Proceedings of the 6th Digital Arts and Culture Conference, IT University of Copenhagen*, 2005, pp. 144–153.

[16] "Cxx-obfus," http://stunnix.com/prod/cxxo/, accessed: 2019-04-28.

[17] "Cobf - the freeware c/c++ sourcecode obfuscator!" https://www.plexaure.de/cms/index.php?id=cobf, accessed: 2019-04-28.

[18] "C source code obfuscator," http://www.semdesigns.com/products/obfuscators/CObfuscator.html, accessed: 2019-04-28.

[19] "Snob - simple name obfuscator," http://www.macroexpressions.com/snob.html, accessed: 2019-04-28.

[20] "String obfuscation system," https://www.codeproject.com/Articles/502283/Strings-Obfuscation-System, accessed: 2019-04-28.

[21] "Obfuscator," https://picheta.me/obfuscator, accessed: 2019-04-28.

[22] "Starforce c++ obfuscator," http://www.star-force.com/products/starforce-obfuscator/, accessed: 2019-04-28.

[23] "Cloakware software protection," https://irdeto.com/software-protection/, accessed: 2019-04-28.

[24] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de) obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, 2006, pp. 140–144.

[25] Zhang, Ouroboros, and jmpews, "Hikariobfuscator - installation," https://github.com/HikariObfuscator/Hikari, accessed: 2019-02-14.

[26] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[27] S. Qing, W. Zhi-yue, W. Wei-min, L. Jing-liang, and H. Zhi-wei, "Technique of source code obfuscation based on data flow and control flow tansformations," in *2012 7th International Conference on Computer Science & Education (ICCSE)*. IEEE, 2012, pp. 1093–1097.

[28] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, p. 4, 2016.

[29] R. Van Riel and G. Vosgerau, *Aussagen-und Prädikatenlogik: eine Einführung*. Springer-Verlag, 2018.

[30] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *International Conference on Information Security*. Springer, 2016, pp. 323–342.

[31] I. V. S. Manoj, "Cryptography and steganography," *International Journal of Computer Applications*, vol. 1, no. 12, pp. 63–68, 2010.

[32] "The llvm compiler infrastructure," https://llvm.org, accessed: 2019-02-18.

[33] J. Singer *et al.*, "Static single assignment books."

[34] "Llvm - the architecture of open source applications," https://llvm.org, accessed: 2019-02-18.

[35] "Llvm language reference manual," https://llvm.org/docs/LangRef.html, accessed: 2019-02-18.

[36] "Llvm clang," http://clang.llvm.org/, accessed: 2019-02-19.

[37] F. Zehender, "Dynamische ausführung von positionstransformationen mittels opengl es 2.0-shaderprogrammen," 2014.

[38] "Llvm backend for mono," https://www.mono-project.com/docs/advanced/runtime/docs/llvm-backend/, accessed: 2019-05-05.

[39] "Llvm - obfuscator wiki," https://github.com/obfuscator-llvm/obfuscator/wiki, accessed: 2019-02-20.

[40] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[41] "Llvm obfuscator - installation," https://github.com/obfuscator-llvm/obfuscator/wiki/Installation, accessed: 2019-02-14.

[42] "Llvm obfuscator - annotations," https://github.com/obfuscator-llvm/obfuscator/wiki/Functions-annotations, accessed: 2019-02-25.

[43] "Hikari introduction," https://github.com/HikariObfuscator/Hikari, accessed: 2019-02-20.

[44] "Hikari wiki," https://github.com/HikariObfuscator/Hikari/wiki, accessed: 2019-05-02.

[45] "Llvm license," http://releases.llvm.org/7.0.0/LICENSE.TXT, accessed: 2019-02-20.

[46] "Llvm obfuscator license," https://github.com/obfuscator-llvm/obfuscator/wiki/License, accessed: 2019-02-20.

[47] "Hikari license," https://github.com/HikariObfuscator/Hikari/wiki/License, accessed: 2019-02-20.

[48] S. Nygard, "Class-dump," http://stevenygard.com/projects/class-dump/, accessed: 2019-02-21.

[49] J. Oliver, C. Cheng, and Y. Chen, "Tlsh–a locality sensitive hash," in *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 2013, pp. 7–13.

[50] "Llvm 4.0.0 clang command line documentation," https://releases.llvm.org/4.0.0/tools/clang/docs/CommandGuide/clang.html, accessed: 2019-02-28.

[51] "Llvm 7.0.0 clang command line documentation," https://releases.llvm.org/7.0.0/tools/clang/docs/CommandGuide/clang.html, accessed: 2019-02-28.

[52] "Llvm's analysis and transform passes," https://releases.llvm.org/7.0.0/docs/Passes.html, accessed: 2019-02-28.