

Obfuskation und Malware Evasion durch Verlagerung von Funktionalität in Threads

Oliver Deja

Matrikelnummer: 6695710

Erstprüfer: Prof. Dr. Jörg Keller

Zweitprüfer: Prof. Dr. Elmar Padilla

Abschlussarbeit im Studiengang

Master of Science Praktische Informatik

eingereicht im August 2021

Selbstständigkeitserklärung

Ich erkläre, dass ich die Abschlussarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung in der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Berlin, den 12. August 2021

Oliver Deja

Inhaltsverzeichnis

1	Einleitung.....	1
2	Grundlagen.....	3
2.1	Begriffsdefinitionen.....	3
2.2	Kontrollflussgraph.....	4
2.3	Reverse Engineering.....	5
2.4	Obfuskation.....	6
2.5	Malware Detektion.....	7
2.6	Malware Evasion.....	9
3	Entwurf und Implementierung.....	12
3.1	Übersicht der Thread-basierten Obfuskation.....	12
3.2	Entwurf.....	14
3.2.1	Entscheidungen und Einschränkungen.....	14
3.2.2	Architektur und Arbeitsweise.....	16
3.3	Implementierung.....	18
3.3.1	Initialisierung.....	18
3.3.2	Datenfluss.....	19
3.3.3	Synchronisation.....	23
3.3.4	Verlagerung.....	23
3.3.5	Besonderheit beim Windows Betriebssystem.....	24
4	Evaluation.....	26
4.1	Beweis der Korrektheit des obfuskierten Programms.....	28
4.1.1	Vergleich zwischen obfuskiertem und nicht obfuskiertem Programm.....	28
4.1.2	Coreutils Testsuite.....	29
4.2	Formelle Obfuskationsmetriken.....	30
4.2.1	Wirksamkeit.....	30
4.2.2	Widerstandsfähigkeit.....	36
4.2.3	Kosten.....	39
4.2.4	Zusammenfassung.....	44
4.3	Auswirkungen auf das Reverse Engineering.....	45
4.3.1	Statische Analyse.....	46
4.3.2	Dynamische Analyse.....	50
4.4	Auswirkungen auf die Malware Detektion.....	51
4.4.1	Signaturbasierte Detektion.....	52
4.4.2	Verhaltensbasierte Detektion.....	55
4.4.3	Heuristische Detektion.....	58
4.4.4	Detektion durch kommerzielle Scanner.....	59
5	Diskussion.....	62
5.1	Vergleich mit ähnlichen Publikationen.....	62
5.2	Ansätze für Gegenmaßnahmen.....	66
5.3	Kombination mit anderen Obfuskationsmethoden.....	67
5.4	Weitere Optimierungen.....	69
5.4.1	Obfuskation.....	70
5.4.2	Funktionsumfang.....	72
5.4.3	Zuverlässigkeit.....	73
6	Zusammenfassung und Ausblick.....	75
7	Anhang.....	77

7.1 Messdaten zur Zunahme der Programmgröße.....	78
7.2 Messdaten zur längsten gemeinsamen Teilsequenz.....	79
7.3 Messdaten zur längsten gemeinsamen Teilzeichenfolge.....	80
7.4 Messdaten zur Zunahme der Instruktionen.....	81
7.5 Messdaten zur Transformationsdauer.....	82
7.6 Messdaten zum Speicherplatz.....	83
7.7 Messdaten zur Performanz.....	84
7.8 Messdaten zur Detektion durch kommerzielle Scanner.....	85
Gliederung der beigelegten CD.....	86
Abkürzungsverzeichnis.....	87
Abbildungsverzeichnis.....	88
Codeausschnittsverzeichnis.....	89
Tabellenverzeichnis.....	90
Literaturverzeichnis.....	91

1 Einleitung

In der Informationstechnologie herrscht ein ständiges „Katz-und-Maus-Spiel“ zwischen Entwicklern (oder auch *Engineers*) auf der einen Seite und *Reverse Engineers* auf der anderen Seite. Dabei ist es unabhängig, ob Entwickler nun kommerzielle oder bösartige Absichten verfolgen beziehungsweise Reverse Engineers versuchen closed-source Software für illegale Zwecke zu analysieren oder versuchen eine automatische Erkennung von maliziöser Software zu realisieren. Eine verbreitete Technik, die Entwickler nutzen, um den Reverse Engineering Prozess zu erschweren, ist die Obfuskation. Unter Obfuskation wird die Veränderung eines (Software-)Produkts unter Beibehaltung der Funktionalität verstanden, mit dem Ziel (böswilligen) Endnutzern unrechtmäßige Angriffe zu erschweren [1]; eine gute Übersicht über die gängigsten Techniken bietet [2].

In dieser Arbeit wird eine Methode zur Obfuskation von Maschinencode vorgestellt und untersucht. Hierbei findet eine Verlagerung zusammenhängender Instruktionen in eigenständige Funktionen statt, welche wiederum als Thread aus der ursprünglichen Funktion aufgerufen werden. Die serielle Ausführungsreihenfolge wird durch entsprechende Synchronisation der einzelnen Threads erreicht.

Das Ziel dieser Technik ist insbesondere die Verschleierung des Kontrollflusses eines Programms, sodass die Effektivität der statischen sowie dynamischen Analyse beeinträchtigt wird. Hinsichtlich der statischen Analyse soll eine Generierung eines Kontrollflussgraphen mit konventionellen Mitteln nicht mehr möglich sein, weil die Ausführungsreihenfolge der einzelnen Threads erst nach Analyse der Synchronisationstechniken bestimmt werden kann. Manuelle Analyseansätze werden durch die zusätzliche Komplexität des Programms erschwert, welche sich durch die Verteilung zusammenhängender Programmteile und der immensen Menge an existierenden Threads auszeichnet.

Überdies findet eine Untersuchung weiterer Wirkungsweisen dieser Transformation statt. Zur empirischen Bewertung der Obfuskation werden allgemeine Metriken wie die Änderung der Performanz oder die Lesbarkeit von transformierten Proben herangezogen, wobei in diesem Kontext bei einer schlechteren Lesbarkeit eine bessere Qualität des Transformationsprozesses zugrunde liegt. Auch Änderungen beim Reaktionsverhalten von automatisierter Analysesoftware zur Erkennung von böartigem Verhalten werden bei der Betrachtung miteinbezogen. In diesem Zusammenhang finden abschließend Überlegungen zu den Möglichkeiten der Rückwandlung eines auf diese Weise obfuskierten Samples statt.

Der Beitrag dieser Arbeit bildet sich aus folgenden Teilen:

- Es wird eine mögliche Implementierung einer Obfuskation vorgestellt, die Parallelität für die Verschleierung des Kontrollflussgraphen nutzt. Diese Implementierung kann als Referenz für ähnliche Vorhaben dienen.
- Die vorgestellte Implementierung wird auf Ihre Leistungsfähigkeit untersucht. Das Schema der Untersuchung kann zudem als Muster für ähnliche Untersuchungen dienen.
- Die vorgestellte Obfuskation wird in einen wissenschaftlichen Kontext gesetzt und die Ergebnisse mit ähnlichen publizierten Arbeiten verglichen.

Die Arbeit ist wie folgt aufgebaut: Das zweite Kapitel dient zur Einführung in die Thematik der Arbeit. Das dritte Kapitel beschreibt den Entwurf und die Implementierung der Obfuskation, inklusive

einer Analyse und Bewertung bei der Umsetzung erkannter Problematiken. Mit Hilfe dieser Implementierung erfolgt im dritten Kapitel die Evaluation der thematisierten Obfuskation erfolgt. Zudem erfolgt an dieser Stelle auch die Betrachtung der Implikationen hinsichtlich der Ergebnisse. Schließlich werden im letzten Kapitel die Evaluationsergebnisse im wissenschaftlichen Kontext eingeordnet und es werden weiterführende Überlegungen angestellt, insbesondere wie auf den Erkenntnissen dieser Arbeit aufgebaut werden kann. Im Übrigen sei angemerkt, dass für englische Fachbegriffe eine deutsche Übersetzung verwendet wurde, insofern die Übersetzung weiterhin ein flüssiges Lesen ermöglicht und ein Rückschluss auf den ursprünglichen Begriff zulässt.

2 Grundlagen

2.1 Begriffsdefinitionen

Da in dieser Arbeit (insbesondere bei der Beschreibung der Softwareimplementierung) Begrifflichkeiten vorkommen, welche im allgemeinen Sprachgebrauch entgegen der gewünschten Bedeutung verstanden werden können, werden diese an dieser Stelle für das bessere Verständnis definiert.

Basic Block: Ein Basic Block bezeichnet eine Abfolge von Maschineninstruktionen ohne Änderung des Programmablaufs (beziehungsweise Kontrollflusses). Kontrollflussänderungen werden durch Sprunginstruktionen, Verzweigungsinstruktionen oder Return-Instruktionen hervorgerufen. Diese Anweisungen beenden somit einen Basic Block.

Funktionalität: Bestandteil einer Funktion im Kontext von Programmiersprachen. Hierbei kann eine einzige (atomare) Instruktion bis zur kompletten Funktion gemeint sein. Entscheidend ist die variierbare **Einteilungsgröße**, die in dieser Arbeit, sofern nicht anders angegeben, ein Basic Block beträgt.

Systemaufruf: Aufruf einer Funktion, die im Kernel-space (also im höher privilegierten Adressraum des Betriebssystems) ausgeführt wird. Für gewöhnlich erfolgt der Aufruf dieser Funktionen durch Funktionen in Systembibliotheken, die durch einen Programmierer über das Application Programming Interface (API) genutzt werden können. Beispielsweise ist auf dem Windows OS die Funktion *CreateThread* eine Bibliotheksfunktion und *NtCreateThread* die durch diese Bibliotheksfunktion aufgerufene Systemfunktion. Da die detaillierte Unterscheidung nicht relevant für das Verständnis ist, wird der Begriff „Systemaufruf“ im Weiteren gleichbedeutend für beide Begriffe verwendet.

Thread: Wird der Begriff „Thread“ in dieser Arbeit verwendet, so ist immer ein Thread im Kontext eines eigenständigen Prozessausführungsfadens, verwaltet durch das Betriebssystem beziehungsweise eine Bibliothek bei Userlevel-Threads, für Multithreadingzwecke gemeint. Dies ist abzugrenzen vom allgemeinen Ausführungs- beziehungsweise Programmfaden (vergleichbar mit dem Kontrollfluss).

Payload: Der Begriff „Payload“ wird gewöhnlich für die Abgrenzung der eigentlichen Nutzdaten von zusätzlich benötigten Informationen (dem sogenannten Overhead) insbesondere bei der informationstechnischen Kommunikation genutzt. Im Kontext von Malware beschreibt dieser Begriff jedoch den gesamten Datenbestand, der die bösartige Software bildet. Dabei kann es sich um (wenige) Maschineninstruktionen oder auch um eine komplette ausführbare Datei handeln.

Aktuelle Funktion, aktueller Basic Block: Die/der in einer Iteration behandelte Funktion / Basic Block.

Während einer Iteration werden auf Basis der aktuellen Funktion neue Funktionen erstellt. Um die neuen Funktionen begrifflich abzugrenzen, werden die folgenden Adjektive hinzugefügt:

Ursprüngliche oder **Aufrufernde** Funktion: Die aktuelle Funktion, deren Ursprungsform dahingehend transformiert wird, dass Basic Blocks in neue Funktionen verlagert werden. Die neuen Funktionen werden durch die ursprüngliche Funktion aufgerufen, deshalb auch „aufrufernd“.

Neue oder **aufgerufene** Funktion: Die neue Funktion, die einen Teil der Funktionalität der alten Funktion übernimmt. Diese neue Funktion wird von der ursprünglichen Funktion durch die Erstellung eines Threads in gewisser Weise aufgerufen, deshalb auch „aufgerufene“.

2.2 Kontrollflussgraph

Maßgeblicher Bestandteil der hier vorgestellten Obfuskation ist die Änderung beziehungsweise Verschleierung des Kontrollflusses eines zu transformierenden Programms. Aus diesem Grund soll an dieser Stelle dieses Prinzip in Verbindung mit der Möglichkeit zur graphischen Darstellung eingeführt werden. Der Kontrollfluss kann auch als Programmablauf bezeichnet werden und erfolgt bei den meisten Instruktionen durch serielle Abarbeitung gemäß ihrer physischen Anordnung im Programm [3]. Das heißt, wenn die Instruktion in einer ersten Zeile abgearbeitet wurde, wird als nächstes die zweite Zeile ausgeführt. Dieser implizite Programmablauf kann durch bestimmte Instruktionen explizit verändert werden. Typisch sind Aufrufinstruktionen, mit denen eine Unterfunktion ausgeführt wird, Return-Instruktionen, welche die Ausführungskontrolle an die Oberfunktion zurückgeben, und Sprunginstruktionen beziehungsweise Verzweigungsinstruktionen, mit denen die Ausführung in einer festlegbaren Zeile (beziehungsweise Adresse bei maschinennaher Betrachtung) fortgesetzt werden kann. Auf weitere Konstrukte, die von Hochsprachen angeboten werden, wird hier nicht eingegangen, da diese Spezialformen im Zusammenhang mit dieser Arbeit nicht relevant sind. Zudem sind die gängigsten Konstrukte durch eine oder mehrere Verzweigungsinstruktionen abbildbar, wie es bei der Umwandlung von der Programmiersprache C/C++ in Maschinencode üblich ist [4]¹.

Da sich der Kontrollflussgraph gewöhnlich auf eine Funktion beschränkt, fließen Aufrufinstruktionen nicht in die Generierung mit ein und anhand von Return-Instruktionen werden nur die Endpunkte einer Funktion festgestellt. Sprunginstruktionen gibt es in bedingter und unbedingter Ausführungsform. Unbedingte Sprunginstruktionen setzen das Programm ohne Ausnahme an der jeweiligen Stelle fort und ähneln einer *goto*-Instruktion. Bei bedingten Sprunginstruktionen findet eine Verzweigung statt (es gibt also mehrere Sprungziele), da die nächste auszuführende Adresse abhängig von einer vorher auszuwertenden Bedingung ist, vergleichbar mit einer *if*-Anweisung. Im Kontrollflussgraph werden diese Sprunginstruktionen nun durch Kantenbildung zwischen zwei Knoten, verkörpert durch zusammenhängende Instruktionen ohne eine den Kontrollfluss beeinflussende Instruktion (ein sogenannter Basic Block), graphisch abgebildet. Durch die Obfuskationstechnik dieser Arbeit werden diese Kontrollflussinstruktionen separiert und in neue Funktionen verschoben, so dass die ursprüngliche Funktion keine einzelne Sprunganweisung mehr beinhaltet und somit keine verwertbare Struktur mehr aufweist.

Zudem wird bei Sprunginstruktionen (und auch bei Aufrufinstruktionen) zwischen „direkt“ und „indirekt“ unterschieden. Bei direkten Sprunginstruktionen ist das Sprungziel, also die nächste auszuführende Instruktion, hartkodiert und deshalb bekannt. Bei indirekten Instruktionen hingegen erfolgt die Berechnung beziehungsweise Bestimmung des Sprungziels erst zur Laufzeit, sodass die indirekten Instruktionen in die Generierung des Kontrollflussgraphen normalerweise auch nicht einbezogen werden. Solche indirekten Instruktionen bilden jedoch die Grundlage für andere Obfuskations-

¹ Neben den drei genannten Instruktionsklassen existieren auf Maschinencodesebene nur zwei weitere Klassen im Zusammenhang mit Interrupts, die den Kontrollfluss beeinflussen (nämlich *IRET* und *RSM*).

methoden, diesbezüglich wird auf den Abschnitt „Kombination mit anderen Obfuskationsmethoden“ verwiesen.

2.3 Reverse Engineering

Reverse Engineering zielt auf die Wiedergewinnung der konkreten Implementierung bei Produkten ab, zu welchen keine Informationen über ihre innere Beschaffenheit vorliegen. [5] Bei Produkten im informationstechnologischen Kontext kann es sich demnach sowohl um Hardware als auch um Software handeln, die Untersuchungen dieser Arbeit beschränken sich jedoch auf letzteres.

Auch hinsichtlich der Form des Softwareprodukts und den Zielen der Analyse existieren weitreichende Unterschiede. Die Software kann als nativer Programmcode vorliegen (also in Maschinensprache), als Bytecode (wie bei Java) oder sogar als Quelltext, wobei ein dokumentierter Quelltext als höchstmögliches Ziel einer rückwärts gerichteten Entwicklung bezeichnet werden kann. Reverse Engineering bei Quelltext ist somit lediglich notwendig, wenn dieser obfuskiert ist, und kann weitestgehend auf den Teilaspekt Deobfuskation (also die Umkehrung der Obfuskation, siehe Abschnitt „Obfuskation“) reduziert werden. Eine weitere häufig angewandte Methode beim Reverse Engineering ist die Dekompilierung, also die Übersetzung der (für den Computer „lesbaren“) ausführbaren Darstellung in eine für den Menschen lesbare Darstellung. Da sich eine Dekompilierung von Bytecode als äußerst einfach und effektiv gestaltet, ist Obfuskation bei solchen Produkten zwingend erforderlich, insofern diese öffentlich zugänglich sind und einer Offenlegung der Funktionsweise vorgebeugt werden soll [2].

Für die Analyse von reinem² Maschinencode ist der meiste Aufwand notwendig. Hierbei wird zunächst das in binärer Darstellung vorliegende Programm mit einem Disassembler in eine textuelle Darstellung gebracht, indem binäre Operationscodes (zum Beispiel 83C002h) durch einen für den Menschen verständlicheren Text (zum Beispiel *add ax, 2*) abgebildet werden. Als nächster Schritt kann die Dekompilierung erfolgen, wobei die Ausgabe von Dekompilern eher eine zusätzliche Abstraktion zur disassemblierten Darstellung bildet und bei steigender Komplexität wenig Parallelen mit der Gestalt des ursprünglichen Quelltextes aufweist [6]. Zielführendere Mittel sind für gewöhnlich die Generierung von graphischen, abstrakten Darstellungen, wie Kontrollflussgraphen und Funktionsaufrufgraphen. Dadurch lassen sich idealerweise Muster wie Schleifen und Funktionalitäten durch die Nutzung bestimmter Systemaufrufe oder ähnlicher Marker erkennen. Hierbei bietet die automatische Analyse durch ein Reverse Engineering Tool Vorteile, insbesondere durch die Auflösung von Adressreferenzen. Neben dieser statischen Analyse kann zudem eine dynamische Analyse mit Hilfe eines Debuggers unterstützen.

In dieser Arbeit wurde das integrierte Reverse Engineering Tool Ghidra [7] verwendet, welches von der National Security Agency als open-source Projekt zur Verfügung gestellt wird und Werkzeuge für die genannten Reverse Engineering Methoden und weitere Funktionalitäten bereitstellt. Ein Überblick über die Software kann in Abbildung 1 gewonnen werden. Dargestellt ist im rechten Fenster der Inhalt des geladenen Programms, also die Operationscodes (kurz Opcodes) mit der entsprechenden textuellen Darstellung inklusive Annotationen, welche bei der automatischen Analyse erstellt werden und zur Unterstützung etwaiger Analysetätigkeiten dienen. Zusätzlich ist im rechten Unterfenster das zugehörige Ergebnis des Decompilers in der Programmiersprache C zu sehen und

² ohne für die Ausführung des Programms unnötige (Debug-)Symbole

im linken Fenster der Kontrollflussgraphen der gesamten Funktion zu erkennen, wobei der gelb markierte Bereich die aktuelle Position des rechten Fensters in der Funktion indiziert.

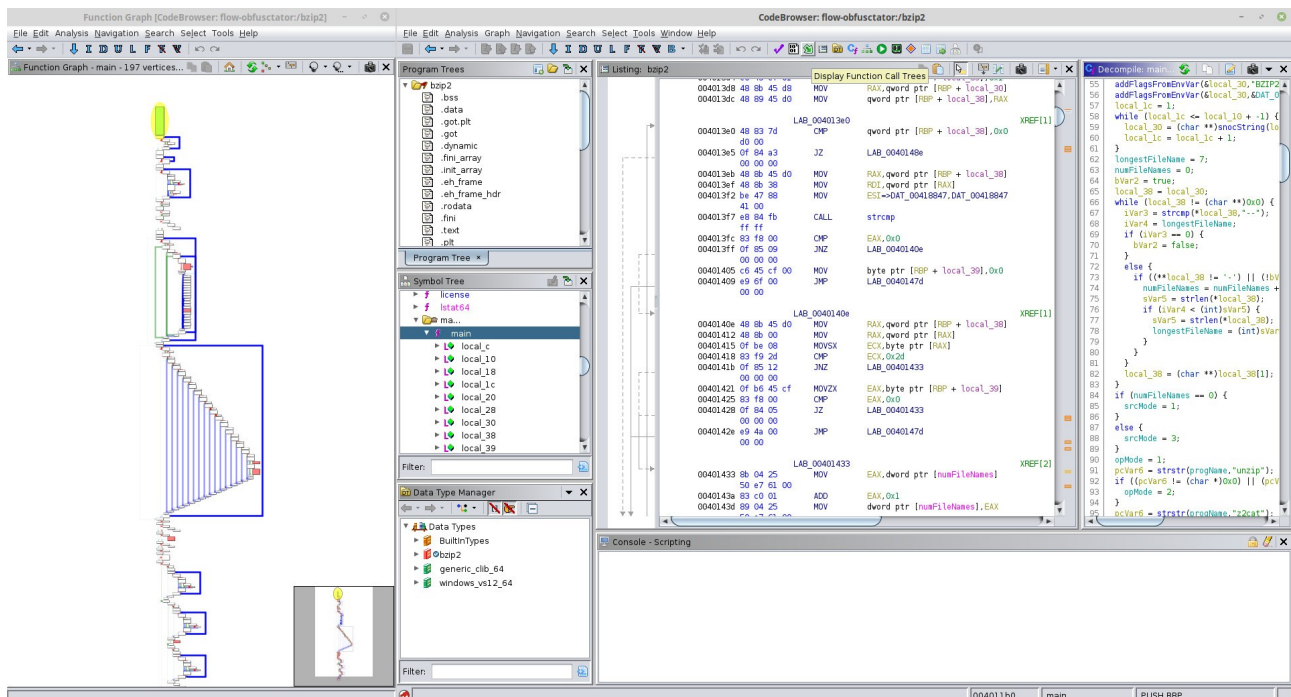


Abbildung 1: Ghidras Benutzeroberfläche

Die Ziele des Reverse Engineering können höchst unterschiedlich ausfallen. Zum einen wird dieses Vorgehen von Sicherheitsanalysten eingesetzt, um existierende Software auf Schwachstellen oder bösartiges Verhalten zu prüfen. Auch Virens Scanner nutzen automatisiert Techniken des Reverse Engineering um malizöse Software zu erkennen. Ebenfalls sind weitere Szenarien mit besonderen Gegebenheiten möglich, wie geringfügige Änderungen an Software, zu welcher der Quellcode abhanden gekommen ist. Zum anderen wird Reverse Engineering des Öfteren auch illegal eingesetzt, um beispielsweise Funktionsweisen proprietärer Software und somit geistiges Eigentum zu extrahieren und für eigene Zwecke zu nutzen oder Modifikationen vorzunehmen, sodass die Notwendigkeit des käuflichen Erwerbs umgangen wird. [8]

2.4 Obfuskation

Um das Reverse Engineering zu erschweren³, kann eine zur Distribution vorgesehene Software durch Transformation obfuskert werden. Hierbei wird zwischen betroffenen Aspekten eines Programms unterschieden, welche sich grob in Anordnung, Daten und Kontrollfluss unterscheiden lassen [2]. Unter Veränderung der Anordnung ist die Veränderung von Funktionshierarchien vorstellbar, eine häufig eingesetzte Methode zur Verschleierung von Daten ist die Verschlüsselung und das Verstehen des Kontrollflusses eines Programms kann durch Einfügen von Maschinenanweisungen ohne Wirkung erschwert werden. Die in dieser Arbeit behandelte Obfuskationsmethode betrifft die Transformation der Anordnung und des Kontrollflusses eines Programms, da zum einen Teile einer Funktion in eine neu erstellte Funktion verschoben werden und zum anderen die Abarbeitung dieser Funktionen anstatt in demselben in mehreren Ausführungsfäden erfolgt.

³ Das Reverse Engineering komplett zu verhindern ist im Prinzip unmöglich, aus diesem Grund wird auch „Security by Obscurity“ (deutsch: Sicherheit durch Unklarheit) als unzureichende Methodik betrachtet.

Wie bereits angedeutet, kann Obfuskation auf unterschiedlichen Abstraktionsebenen erfolgen. Einerseits ist die Transformation des Quellcodes eines Programms denkbar, wie in [9] angewandt. Dies erfordert eine lexikalische Analyse des jeweiligen Quellcodes, demnach müsste für jede zu unterstützende Sprache ein eigenes Frontend für diese Analyse implementiert werden. Als andere Möglichkeit sei die Anwendung der Transformation auf die ausführbare Darstellung genannt. Auch hier ist die Implementierung unterschiedlicher Frontends für die verschiedenen Darstellungen notwendig, wie bei nativem Maschinencode, Java- oder .NET-Bytecode. Zudem ist die Modifikation von nativen ausführbaren Dateien (Binary Rewriting) mit einigen Schwierigkeiten behaftet [10]. Die generischste Lösung ist die Modifikation von Programmabläufen in einer Zwischensprache. Jedoch variieren auch hier die Zwischensprachen zwischen verfügbaren Compilern. Dementsprechend erfolgt eine Festlegung auf einen bestimmten Compiler und die von diesem unterstützten Ein- und Ausgabesprachen. Je nach Zielsystem⁴ können zusätzlich Einschränkungen hinzukommen oder es kann die Notwendigkeit zu weiteren Implementierungen entstehen.

Als Gegenmaßnahme durch sogenannte Deobfuskation kann die Normalisierung von Code jeglicher Form ein probates Mittel sein. Hierzu können sogar herkömmliche Compileroptimierungen eingesetzt werden, weil die damit verbundene Reduktion und Simplifikation des Codes eine der Obfuskation entgegengesetzte Wirkung entfalten kann [11]. Bei komplexeren Obfuskationsmechanismen ist die Kenntnis des Algorithmus erforderlich, welcher gegebenenfalls erst mit Reverse Engineering Methoden rekonstruiert werden muss.

2.5 Malware Detektion

Der Bedarf an Erkennung von bösartiger Software (oder kurz Malware Detektion) ist im geschäftlichen sowie privaten Umfeld weit verbreitet. Das Windows Betriebssystem ist als Zielplattform für Entwickler solcher Software aufgrund der starken Verbreitung als Desktop PC besonders attraktiv, so wird das Windows OS auch standardmäßig mit einem Virens Scanner ausgeliefert. Zudem geraten die Betriebssysteme für mobile Endgeräte aufgrund ihrer Beliebtheit stärker in den Fokus von Angreifern. [12]

Zur Erkennung von Malware werden (neben netzwerkbasierten Lösungen) Malware Scanner beziehungsweise Virens Scanner direkt auf dem zu schützenden Gerät installiert und eingesetzt. Solche Virens Scanner setzen verschiedene Techniken ein, um die Sicherheit des Geräts zu gewährleisten. Neben der Analyse einer Datei selbst und eines Prozesses während der Ausführung ist auch die Prüfung von (plattformabhängigen) Systemeinstellungen⁵ ein zielführendes Vorgehen. Ebenfalls kann die Analyse manuell mit Hilfe von Reverse-Engineering-Techniken erfolgen. [13]

Da informationstechnische Angriffe über verschiedene Vektoren erfolgen können und über diese Vektoren zudem unterschiedliche bösartige Payloads eingesetzt werden können, ist die Abdeckung eines breiten Spektrums durch einen Virens Scanner erforderlich. Mögliche Angriffsvektoren sind beispielsweise die Übermittlung von bösartiger Software über Mail oder die Ausnutzung von Schwachstellen in auf dem Gerät verwendeter Software wie einem Internetbrowser. Die Gestalt der maliziösen Payloads ist nahezu unbegrenzt und reicht von eingebettetem JavaScript in PDF-Dateien oder

4 zum Beispiel Betriebssysteme (Windows, Linux, *et cetera*) oder auch höhere Ausführungsumgebungen wie die Java Virtual Machine

5 zum Beispiel für den automatischen Start beim Bootvorgang des Geräts registrierte Applikationen

Makros in Microsoft Office Dokumenten, über ausführbaren Code, welcher von der Software abhängig ist, in welche dieser injiziert wird, bis zum reinen nativen Maschinencode (unabhängig ob als Shellcode⁶ oder als ausführbare Datei). Da über Maschinencode die bestmögliche Kontrolle eines Zielsystems erreicht werden kann, ist der vielzählige Einsatz dieser Form naheliegend. So ist es üblich, dass eine vom Angriffsvektor abhängige Payload zur Infektion des Ziels die eigentliche bösartige Software in Form von Maschinencode nachlädt und verbringt. Dementsprechend kann angenommen werden, dass bei der Entwicklung von Virenscannern die Erkennung von bösartigem Maschinencode hoch priorisiert wird.

In der Literatur gibt es verschiedene Ansätze, die Techniken zur Detektion von Malware zu unterteilen. Im Groben lassen sich diese Techniken in drei Kategorien unterteilen: signaturbasierte, verhaltensbasierte und statistische (oder heuristische) Detektion [14].

Die gängigste Methode ist die Detektion durch Signaturen. Hierbei wird für die Malware oder ein Teil der Malware ein möglichst eindeutiger Wert abgeleitet, über welchen die Software identifiziert werden kann. Der Einsatz von Hash-Algorithmen für die Erzeugung eines solchen Werts ist weit verbreitet. Da jedoch die Modifikation eines einzigen (für die Funktion der Malware unerheblichen) Bits den Hash-Wert nicht mehr wiedererkennbar verändert, wurden Lokal Sensitive Hash-Funktionen entwickelt, die einen Ähnlichkeitsvergleich ermöglichen [15, 16]. Eine relativ neue Form der Signaturerstellung bietet die Software YARA [17], welche auf der Mustererkennung auf Byteebene durch reguläre Ausdrücke basiert. Auch weitere weniger veränderbare Eigenschaften von Software lassen sich für die Erzeugung von Signaturen nutzen, wie Kontrollflussgraphen [18, 19, 20, 21] oder Basic Blocks [22]. Ein Nachteil der signaturbasierten Methode ist, dass eine beträchtliche Menge an Signaturen für den Abgleich in einer Datenbank vorliegen müssen und die Detektion bei Fehlen der Signatur fehlschlägt.

Zudem reicht dieser Ansatz nicht, um sich stark verändernde oder neuwertige Malware zu erkennen. Hierfür wird durch Virenscanner das Verhalten einer Malware analysiert. Dabei gibt es zahlreiche Aspekte, die bei der Betrachtung herangezogen werden können, unter anderem die Abfolge von Systemaufrufen. Diese Untersuchung kann sowohl statisch als auch dynamisch erfolgen. Ein großer Nachteil der statischen Analyse liegt in der Eventualität, dass die zu untersuchenden Daten verschlüsselt vorliegen und erst während der Ausführung entschlüsselt werden. Zudem ist die Abdeckung jegliches Verhaltens durch statische Analyse gemäß dem Halteproblem als unentscheidbar eingestuft [23]. Aus diesem Grund erfolgt die Untersuchung des Verhaltens von Software dynamisch, also durch direkte Ausführung in einem isolierten System (auch Sandbox genannt) häufig auf Basis von Virtualisierungstechniken. Der Nachteil bei der dynamischen Analyse ist jedoch der Ressourcenaufwand, der üblicherweise auf einem End- beziehungsweise Live-System nicht zur Verfügung steht. Deshalb bieten Hersteller von Antivirus-Produkten häufig eine Cloud-basierte Unterstützung an, sodass zu analysierende Dateien auf die Server des Anbieters hochgeladen und dort dynamisch untersucht werden [24]. Falls die entsprechenden Ressourcen vorhanden sind, kann solch eine Untersuchung auch durch Einsatz von open-source Produkten (zum Beispiel Cuckoo Sandbox [25]) erfolgen. Die Analyse von sich in Ausführung befindlichen Prozessen auf einem Live-System wird ebenfalls häufig angewandt. Neben der Überwachung der Systemaufrufe und de-

6 Shellcodes sind reine Maschineninstruktionen in binärer Darstellung, die gewöhnlich durch Schwachstellen in Software unberechtigt zur Ausführung gebracht werden und einem Angreifer häufig den Zugriff auf ein System über einen Kommandozeileninterpreter (auch „Shell“ genannt) ermöglichen sollen.

ren Wirkung ist diesbezüglich auch die Erkennung von verdächtigen Eigenschaften wie Speicherbereichen mit unüblichen Berechtigungsmustern gängig. Ebenso kann die signaturbasierte Detektion hier Anwendung finden, was insbesondere bei verschlüsseltem oder nachgeladenem Code notwendig ist. Bei einer derartigen Detektion kann jedoch eine Kompromittierung des Systems nicht mehr ausgeschlossen werden.

Ein weiterer Ansatz wird durch statistische beziehungsweise heuristische Methoden erreicht und kann äußerst unterschiedliche Ausprägungen annehmen, unter anderem durch Kombination der zuvor genannten Ansätze. Aktueller Forschungsgegenstand ist insbesondere die Anwendung von Mitteln aus dem Bereich des maschinellen Lernens [26, 27, 28, 29, 30]. Auch Hersteller von Virensclannern werben mit dieser Fähigkeit für ihr Produkt [31, 32]. Ein weniger komplexer Ansatz ist die Erkennung von sich selbst entpackenden beziehungsweise entschlüsselnden ausführbaren Dateien und der Einleitung von Folgemaßnahmen, wie die statische Extraktion des kodierten Bereichs oder die dynamische Analyse. Ferner wird die automatische Erkennung von vermeintlich obfuskierten Daten eingesetzt. Hierbei besteht jedoch die Gefahr, dass gutartige Software fälschlicherweise als Malware eingestuft wird, wodurch das Benutzererlebnis negativ beeinflusst werden kann. Sobald auf Basis von komplexen Signaturen wie Kontrollflussgraphen oder Basic Blocks weiterführende heuristische Mittel angewandt werden, sind diese ebenfalls eher in dieser Kategorie zu verorten. Anspruchsvollere Methoden nutzen Verhaltensanalyse, um daraus fein granulare Merkmale abzuleiten, die als Basis für maschinelles Lernen dienen [27]. Ein weiterer Vorteil neben der höheren Erkennungsrate von Malware ist die mögliche Beschleunigung des Analysevorgangs beziehungsweise Reduzierung des benötigten Rechenaufwands. So wirbt Kaspersky mit einem zweistufigen Analysedesign, das eben diese Optimierung verspricht [31]. Eine Vielzahl wissenschaftlicher Untersuchungen behandelt zudem Spezialfälle hinsichtlich Eigenschaften bestimmter Virustypen oder der robusten Detektion bei speziellen Taktiken zur Vermeidung der Detektion (oder auch Malware Evasion genannt) [11, 19, 22, 33, 34].

2.6 Malware Evasion

Malware Evasion [35] (deutsch: Ausweichen der Malware) bezeichnet die Kunst bösartiger Software einer Entdeckung zu entgehen. Bewusst wurde hier der Begriff „Kunst“ gewählt, da Möglichkeiten und Ideen in dieser Hinsicht keine Grenzen gesetzt sind. Auf der einen Seite kann die Methode möglichst allgemein gehalten werden, auf der anderen Seite kann sie für eine bestimmte Detektionsmethode oder ein bestimmtes Zielsystem zugeschnitten sein. In der Literatur fokussieren sich die Methoden auf netzwerkbasierte Techniken, um die Malware an einem Intrusion Detection System vorbeizuschleusen. Die bekanntesten Kategorien beziehungsweise Methoden seien im Folgenden aufgeführt (wobei sich diese teilweise überschneiden):

- Umgebungssensitivität: Erkennung eines an den Prozess gehängten Debuggers oder einer Sandbox als Ausführungsumgebung [36].
- Verschlüsseln oder Packen von Daten und das Entschlüsseln beziehungsweise Entpacken während der Laufzeit (mit verfügbaren Werkzeugen wie UPX⁷ [37] oder individuell).

⁷ Mit UPX gepackte ausführbare Dateien werden von den meisten Virensclannern bereits ohne nähere Untersuchung als verdächtig eingestuft, obwohl die Dateien auch gutartig sein können.

- Rückwärtsorientierte Programmierung (oder Code-Wiederverwendung): Bei bekannter Datenanordnung eines Ziels die Nutzung von Code eines infizierten beziehungsweise zu infizierenden Prozesses [38].
- Adversarial Attacks: Ausnutzung von Störungen im Kontext des maschinellen Lernens (Ziel-spezifisch) [39].
- Mimikry: Das Vortäuschen von gutartigem Verhalten, um die böartigen Handlungen zu verbergen. Zum einen kann durch eine Vielzahl an normalen Aktionen die Anzahl von verdächtigen Aktionen relativiert werden. Zum anderen können maliziöse Aktionen so durchgeführt werden, dass diese gutartig erscheinen [40].
- Obfuskation: Veränderung des Programmcodes, sodass Detektionsmethoden, die sich Reverse Engineering Techniken bedienen, beeinträchtigt werden.
- Polymorphie (oder auch Vielgestaltigkeit): Wandlung der Form einzelner Bestandteile oder (falls möglich) der kompletten Erscheinung von Software unter Beibehaltung der Funktionalität.

In der Literatur wird zwischen Polymorphie und Metamorphie unterschieden [41]. Wohingegen Polymorphie meist auf die Veränderung durch Verschlüsselungsmechanismen mit mutierenden Parametern reduziert wird, fokussiert sich Metamorphie auf die Transformation des eigentlichen Maschinencodes. Diese beiden Techniken werden oft in Verbindung mit teilweise reflexiven Techniken gebracht, das heißt es wird erwartet, dass sich die Software *selbst* verändert und mutiert [11, 19, 33, 41, 42]. Dies kann auf den Umstand zurückzuführen sein, dass diese Begriffe und Definitionen in wissenschaftliche Publikationen als Reaktion auf diese einst neuartige Form von Viren entstanden sind, ohne die Verteidigungsstrategien generalisiert auf die Diversität von Malware zu beziehen. In dieser Arbeit werden diese Restriktionen zur Vereinfachung aufgehoben und damit die Bedeutung des Wortes Polymorphie annähernd mit dem Begriff „Softwarediversität“ gleichgesetzt.

Obfuskation und Polymorphie werden öfters in Verbindung gebracht, weil randomisierte Obfuskation eine Methode zur Erzeugung von Polymorphie sein kann [43]. So wurden bereits Konzepte vorgestellt, die Transformationen (welche zu den Obfuskationstechniken gezählt werden können) mit zufälligen Parametern während der Kompilierung vornehmen, um Malware zu diversifizieren. In [44] wurde in diesem Zusammenhang proklamiert, dass mit dieser Methode die signaturbasierte und ähnlichkeitsbasierte Analyse als nicht mehr effektiv betrachtet werden kann. Eine Untersuchung, wie Scanner auf diese Technik reagieren, fand jedoch nicht statt. In [9] wurde überdies das maschinelle Lernen in die Generierung einer nicht detektierbaren Malware einbezogen; ein weiterer Generator wurde in [45] vorgestellt, jedoch ohne Bezug zur Malware Evasion. Dieses Konzept der Softwarediversität wird zugleich als Gegenmaßnahme von Malware Evasion, im Speziellen bei der rückwärtsorientierten Programmierung, genutzt, da hierbei die Datenanordnung des Ziels bekannt sein muss, die Diversifikation dem Angreifer dieses Wissen jedoch entzieht [46].

Es gibt etliche Maßnahmen, um Malware Evasion Techniken entgegenzuwirken. Neben der bereits erwähnten Softwarediversität, welche samt Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) und Kontrollfluss-Guards, die Nutzbarkeit bestimmter Angriffsvektoren verhindern soll, ist die Deobfuskation ein gängiges Mittel, um eine variierte Malware Form für eine bessere Verarbeitung (gegebenenfalls mit Detektion von böartigem Verhalten) zu kanonisieren.

Auch wissenschaftliche Ansätze, die beispielsweise auf der Generierung weniger veränderlicher Signaturen aufsetzen (wie Kontrollflussgraphen), zielen konkret auf die Erkennung polymorpher Viren ab. Einen Einstieg in diesen Bereich bietet [47].

3 Entwurf und Implementierung

In diesem Abschnitt wird nach Beschreibung des Entwurfs, der dabei getätigten Entscheidungen und der für die Arbeit gesetzte Beschränkungen eine konkrete Implementierung der Thread-basierten Obfuskation vorgestellt. Die hier beschriebene Implementierung stellt eine mögliche Lösung dar, bei der selbstverständlich Teilaspekte durch Alternativen substituierbar sind. Einige offensichtliche oder aussichtsvolle Alternativen werden im Kapitel „Diskussion“ unter Beleuchtung der Implikationen vorgestellt. Um die Übersichtlichkeit zu gewährleisten, konzentriert sich die praktische Evaluation dieser Arbeit jedoch nur auf die in diesem Kapitel beschriebene Lösung.

3.1 Übersicht der Thread-basierten Obfuskation

Dieser Abschnitt dient der Einführung in das Konzept des für diese Arbeit erstellten Programms. Wie bereits in der Einleitung dargelegt, soll mit dieser Arbeit eine konkrete Obfuskationstechnik vorgestellt, implementiert und untersucht werden. Diese Technik hat die Verschleierung des Kontrollflussgraphen von Funktionen zum Ziel, indem die Instruktionen einer Funktion in getrennten Threads ausgeführt werden. Um die serielle Ausführungsfolge einzuhalten, wird eine entsprechende Synchronisation der Threads realisiert. Zum Verständnis soll der Unterschied zwischen der nicht obfuskerten Darstellung (Codeausschnitt 1) und der obfuskerte Darstellung (Codeausschnitt 2) unter Zuhilfenahme eines vereinfachten C-Quellcodes [48]⁸ veranschaulicht werden.

```
int main(int argc, char *argv[]) {
    fCreateToolhelp32Snapshot = GetProcAddress(kernel32_dll, s_CTS);
    fProcess32First = GetProcAddress(kernel32_dll, s_PF);
    fProcess32Next = GetProcAddress(kernel32_dll, s_PN);
    fOpenProcess = GetProcAddress(kernel32_dll, s_OP);

    hProcess = fCreateToolhelp32Snapshot(TH32CS, 0))
    pe32.dwSize = sizeof(PROCESSENTRY32);
    fProcess32First(hProcess, &pe32))

    while (fProcess32Next(hProcess, &pe32)) {
        if (!strcmpi(pe32.szExeFile, inj_proc)) {
            hProc = fOpenProcess(PROCESS_VM_READ, FALSE, procID)
            break;
        }
    }
}
```

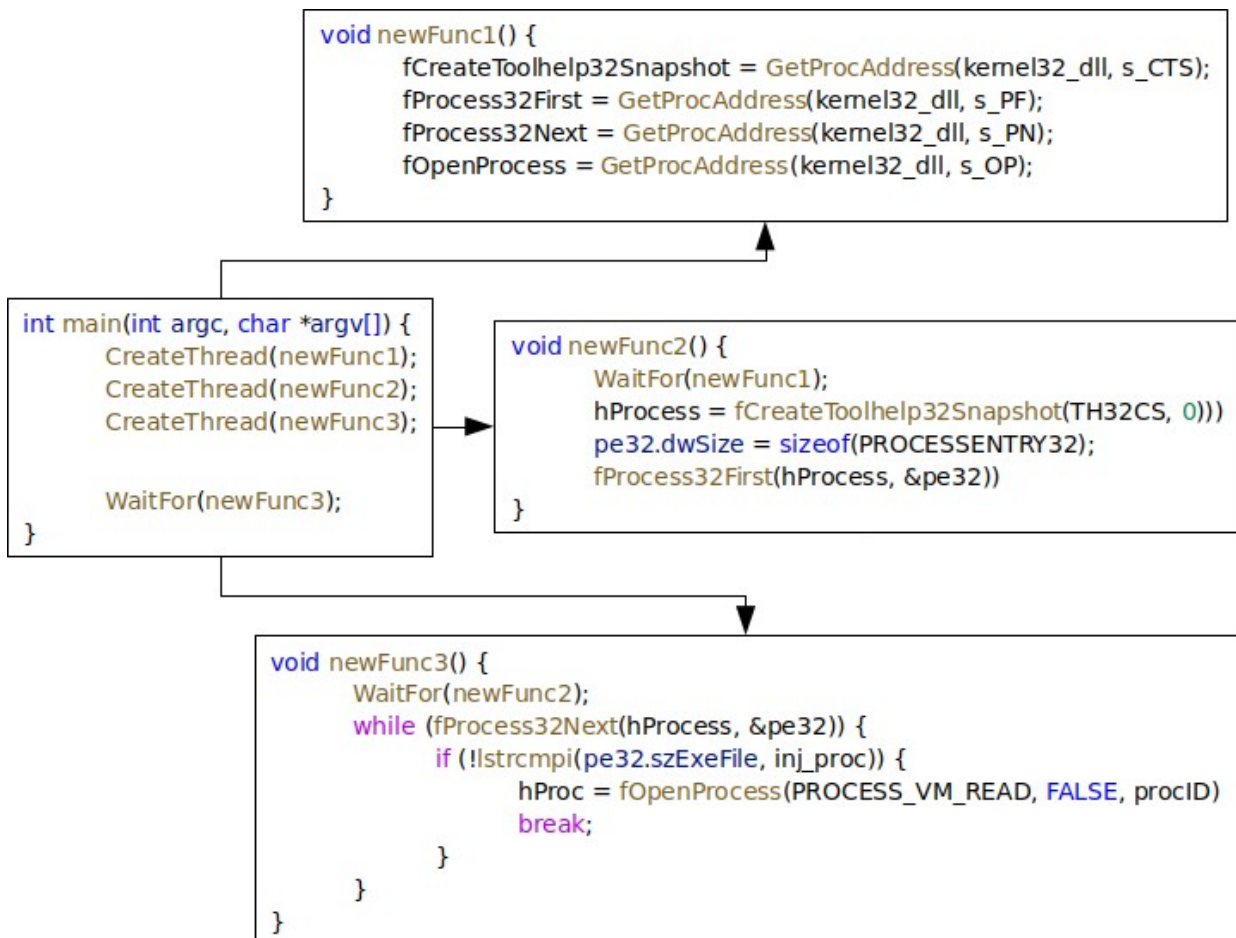
Codeausschnitt 1: Nicht obfuskerte Funktion

Im Codeausschnitt 1 ist eine *main*-Funktion dargestellt, welche in drei Bereiche unterteilt wurde (getrennt durch eine Leerzeile). Für jeden dieser Bereiche wird eine neue Funktion kreiert, nämlich *newFunc1*, *newFunc2* und *newFunc3*. Jede dieser Funktionen wird durch die ursprüngliche Funktion gemäß Codeausschnitt 2 als Thread aufgerufen. Die Einhaltung der Ausführungsfolge erfolgt, indem am Anfang jeder neuen Funktion (mit Ausnahme der ersten) gewartet wird, bis der Vorgänger die Ausführung beendet hat. Die ursprüngliche Funktion wartet, bis die letzte aufgerufene Funktion

8 malwares/Source/Original/XOR-USB_Jan2009.zip → main.c

die Ausführung beendet hat, um eine Prozessterminierung beziehungsweise ein Return der Funktion vor Abarbeitung der Funktionalität zu verhindern.

Verschiedene Implikationen lassen sich anhand des Beispiels bereits erkennen. Zum einen muss neben der Verlagerung der Instruktionen der Datenfluss umstrukturiert werden, sofern die benutzten Variablen nicht im globalen Raum deklariert wurden⁹. Zum anderen lässt sich bereits die Formveränderung des Kontrollflussgraphen antizipieren, der allein aus der ursprünglichen Funktion nicht mehr abgeleitet werden kann und sich nun über drei einzelne Funktionen streckt. Mit feinerer Unterteilung der ursprünglichen Funktion wird zudem eine ausgeprägtere Wirkung durch die neu erstellten Funktionen erzielt.



Codeausschnitt 2: Obfuskierte Funktion

Im Folgenden wird die Obfuskation auch kurz als „Thread-basierte Obfuskation“ bezeichnet.

⁹ Hinweis: In der Veranschaulichung wurde auf eine Deklaration der Variablen für eine kompakte Darstellung verzichtet.

3.2 Entwurf

3.2.1 Entscheidungen und Einschränkungen

Rahmenwerk

Wie bereits im Abschnitt „Obfuskation“ erläutert, kann die Transformation auf unterschiedliche Darstellungsformen von Code angewandt werden. Zudem wurde im Abschnitt „Malware Detektion“ begründet, dass als Zielcode die native Form zweckmäßig ist. Um die Lösung möglichst generisch zu realisieren und die Komplexität (die sich insbesondere beim Binary Rewriting potenzieren würde) gering zu halten, wurde sich in dieser Arbeit auf die Transformation auf Basis einer Zwischensprache während der Kompilierung festgelegt, wie bei einigen wissenschaftlichen Publikationen, die ähnliche Ziele verfolgen [10, 44, 45, 49, 50]. Ebenfalls soll die open-source LLVM Compiler Infrastruktur [51] verwendet werden, da sich dieses Projekt hinsichtlich seiner Bekanntheit, handhabbaren Modularität und befriedigenden Dokumentation von den Alternativen abhebt. Die Zwischensprache des Compilers, auf welcher die Implementierung basieren soll, nennt sich LLVM Intermediate Representation (LLVM IR).

Im Weiteren erfolgen im Rahmen dieser Arbeit Einschränkungen bezüglich des Frontends, also der Quellsprache, und des Backends, also der Zielplattform.

Zielbetriebssystem

Da als Ergebnis des Obfuskationsprozesses nativer Maschinencode emittiert werden soll, muss an dieser Stelle neben den durch den gewählten LLVM Compiler eingeschränkten Zielprozessoren das Zielbetriebssystem festgelegt werden. Die Festlegung auf ein beziehungsweise mehrere Zielbetriebssysteme ist erforderlich, weil Parallelisierungsmechanismen genutzt werden sollen und diese (beziehungsweise die Bibliotheken, die diese Mechanismen bereitstellen) betriebssystemabhängig sind. Zwangsläufig müssen entsprechende Parallelisierungsmechanismen auf dem Zielbetriebssystem verfügbar sein.

Als Zielbetriebssysteme ist zum einen das Windows OS vorgesehen, weil dieses aufgrund der starken Verbreitung [52] ein begehrtes Ziel von Angreifern ist und dementsprechend eine zielorientiertere Evaluation gegenüber etwaigen Alternativen zu erwarten ist. Als weiteres Zielbetriebssystem wurde das Linux OS gewählt, da open-source Software auf Linux eine stärkere Unterstützung erfährt und die Implementierungsarbeit in diesem Kontext erleichtert. Dies wurde während der Implementierungsarbeit bestätigt, weil Plugins mit dem neuen LLVM Pass Manager im Gegensatz zu Linux auf Windows nicht funktionieren und bei der Nutzung des alten Pass Managers zusätzliche Maßnahmen nötig sind [53], sodass der Aufwand bei der Arbeit mit dem Windows OS erhöht wird. Die fertige Lösung kann anschließend mit minimalen Anpassungen für Windows portiert werden. Da auf Ebene der LLVM IR keine signifikanten Unterschiede zwischen der 32-bit und 64-bit Architektur bestehen, sollen beide Ziele unterstützt werden. Das LLVM Projekt wurde in der Programmiersprache C++ entwickelt. Die hinsichtlich der Zielplattformen abweichende Anwendung bestimmter Transformationsschritte wird in der Implementierung durch entsprechendes C-Preprocessing gelöst.

Quellsprache

Als Nächstes erfolgt eine Einschränkung hinsichtlich der Quellsprache. Hierbei können lediglich Programmiersprachen betrachtet werden, die in der LLVM Compiler Infrastruktur unterstützt werden und im Backend in nativen Code übersetzt werden können. Da offizieller Support vom LLVM Projekt lediglich für C/C++ durch den Clang [54] Compiler existiert und sich diese Sprachenfamilie entsprechend ihrer andauernden Beliebtheit [55] trotz hohen Alters bewährt hat, werden diese Sprachen im weiteren Verlauf dieser Arbeit als Quellsprache herangezogen.

Parallelisierungsmechanismen

Parallelisierung soll durch die entsprechenden Aufrufe der Betriebssystem API erreicht werden. Bei Windows erfolgt das Starten eines Threads beispielsweise über die Funktion *CreateThread*, wohingegen der Name der vom Linux OS exportierten Funktion *pthread_create* lautet. Dies ließe sich gegebenenfalls durch den Einsatz der C++ Standard Bibliothek generalisieren, sodass lediglich der Konstruktor der Klasse *thread* aufgerufen werden müsste. Allerdings entsteht durch den Einsatz der Standardbibliothek ein nicht zu vernachlässigender Overhead, insbesondere durch C++-inhärente Funktionen wie die automatisierte Verwaltung des dynamischen Speichers und Ausnahmebehandlungen, die durch den Algorithmus berücksichtigt werden müssten (vergleiche fünf Codezeilen mit einundzwanzig Codezeilen in der LLVM IR bei Übersetzung von einer Instruktion zur Erstellung eines Thread gefolgt von einer Instruktion zur Synchronisation mit besagtem Thread). Folglich werden die Systemaufrufe für die Implementierung dieses Aspekts genutzt.

Anpassung des Datenflusses

Da der Datenfluss innerhalb einer Funktion gewöhnlich direkt über lokale Variablen auf dem Stack realisiert wird [56] und die genaue Position einer lokalen Variable im Normalfall nur der zugehörigen Funktion bekannt ist, müssen Anpassungen erfolgen, damit die neuen Funktionen auf gemeinsam genutzte Variablen zugreifen können. Zum einen besteht die Möglichkeit, dass die aufrufende Funktion die benötigten Speicherbereiche auf dem Stack oder Heap allokiert und die Zeiger als Argumente an die aufgerufene Funktion übergibt. Zum anderen können lokale Variablen auch in den globalen Kontext verschoben werden, sodass jede Funktion über ihre eindeutige Adresse auf diese zugreifen kann. Da die letztere Möglichkeit deutlich weniger Verwaltungsaufwand verursacht und folglich die Implementierung vereinfacht sowie die Ausführungsdauer weniger stark erhöht, wurde sich für die Nutzung von globalen Variablen entschieden.

Einteilungsgröße

Die Feinheit bei der Unterteilung einer ursprünglichen Funktion in auszulagernde Bereiche und somit die Anzahl an neu zu erstellenden Funktionen lässt sich beliebig wählen und könnte auch zur Realisierung eines Polymorph-Generators randomisiert werden. Bei der Implementierung wurden Basic Blocks als auszulagernde Einheit gewählt, da hierdurch die Komplexität beherrschbarer ist und diese Einteilung als konzeptioneller Beweis ausreichend ist. Dass kleinere Einteilungen auch möglich sind, kann induktiv durch folgende Überlegung bewiesen werden: Wenn ein beliebiger Basic Block b bei der Transformation als Einteilungsgröße gewählt werden kann und sich ein Basic Block durch Einfügen einer bedingungslosen Sprunginstruktion bei mindestens zwei Instruktionen in zwei Basic Blocks b_1 und b_2 aufteilen lässt, so können Basic Blocks mit einer minimalen Größe von einer Instruktion gebildet und bei der Transformation als Einteilungsgröße gewählt werden. Der

induktive Beweis hinsichtlich einer größeren Einteilung erfolgt *vice versa* und ist durch die maximale Anzahl an Instruktionen in einer Funktion beschränkt.

Besonderheit: rekursive Funktionen und Funktionen mit variabler Parameteranzahl

In der Implementierung der Arbeit werden rekursive Funktionen nicht transformiert, da der ausgearbeitete Algorithmus nicht dafür vorgesehen ist. Aufgrund des Umstands, dass jede lokale Variable in genau *eine* neue globale Variable umgewandelt wird, würde eine rekursiv aufgerufene Funktion n die globalen Variablen der rekursiv aufgerufenen Funktion $n-1$ überschreiben. Ebenfalls werden Funktionen mit einer variablen Parameteranzahl (wie bei *printf*) von der Transformation ausgenommen, weil hierbei für jeden (lokalen) Parameter genau eine globale Variable erstellt werden müsste und dies bei einer unbekannten Parameteranzahl nicht möglich ist. Als mögliche Lösung für diese beiden Fälle wäre der Einsatz einer dynamischen Listenstruktur denkbar.

Besonderheit: Ausnahmebehandlung bei C++

Überdies entstehen Schwierigkeiten bei der Verlagerung von Funktionalität, die durch das Ausnahmebehandlungskonzept der Sprache C++ abgedeckt sind. Insofern sich Funktionen in einem klassischen *try-catch*-Block befinden, erfolgt der Aufruf dieser Funktionen unter Angabe desjenigen Basic Blocks, welcher bei Eintreten einer Ausnahme als Nächstes ausgeführt werden soll. Dieser Basic Block ist zunächst für die Bereinigung zuständig, ruft also Destruktoren auf und steuert die weitere Abwicklung der Ausnahme. Ist bei der Abwicklung die Funktion mit der *catch*-Klausel erreicht, so wird der dort definierte Benutzercode ausgeführt und das Programm wird ordentlich fortgesetzt. Andernfalls wird der Bereinigungsprozess in den zuständigen Basic Blocks der aufrufenden Funktionen fortgesetzt. An dieser Stelle tritt das Problem auf, dass aufgrund der durch die Threads kreierten Schicht zwischen Entstehungsstelle der Ausnahme und der endgültigen Behandlungsroutine die modellierte Verbindung verloren geht. Zudem existieren Unterschiede bei der Ausnahmebehandlung hinsichtlich der Zielsysteme Linux und Windows. Dieses Thema soll aufgrund der Komplexität hier nicht näher ausgeführt und stattdessen auf die offizielle LLVM Dokumentation [57] verwiesen werden. Im Weiteren wird für diese Arbeit vorausgesetzt, dass keine Ausnahmebehandlungen im Quellcode eingesetzt oder Funktionen mit Ausnahmebehandlung bei der Transformation ausgelassen werden.

3.2.2 Architektur und Arbeitsweise

Nachdem die Entwurfsentscheidungen getroffen wurden und die entsprechenden Ziele für die Implementierung der Arbeit gesetzt wurden, soll an dieser Stelle die Architektur sowie die Arbeitsweise der Obfuskation vorgestellt werden.

Wie aus den Entwurfsentscheidungen hervorgeht, wird als Rahmenwerk die LLVM Compiler Infrastruktur verwendet. Die Modifikationen für die Realisierung der Obfuskation sollen ähnlich einer Compileroptimierung erfolgen. Hierfür wird ein entsprechender Algorithmus in der Programmiersprache C++ unter Nutzung von Klassen und Funktionen des LLVM Rahmenwerks erstellt. Um diesen Algorithmus für die Umwandlung einzusetzen, existieren zwei Möglichkeiten [57]:

1. Erstellen einer Shared Library als Plugin, welche dynamisch durch den modularen LLVM IR Optimierer *opt* geladen wird und die gemäß Schnittstellenvorgaben exportierte Funktion in dieser Library ausführt. Vorteil: Nicht das gesamte LLVM Projekt muss eigenständig

kompiliert werden, sondern vorkompilierte ausführbare Dateien können bezogen werden. Nachteil: Der Quellcode muss zuerst durch ein Frontend in die LLVM IR übersetzt werden, was den Build-Prozess verkompliziert.

2. Einfügen des Quellcodes der Obfuskation in den Quellbaum des LLVM Projekts, sodass die Transformation in den Kompilaten integriert ist. Vorteil: Die Obfuskation kann direkt während des Kompilervorgangs des zu transformierenden Programms erfolgen. Nachteil: Es sind einige (teilweise komplexe) Anpassungen in mehreren Dateien des LLVM Projekts erforderlich.

Für die Arbeit wurden beide Optionen umgesetzt. Die erste Option ist bei der Entwicklung vorteilhaft, da die Kompilierdauer der Shared Library gering ist und somit Änderungen am Quellcode zügig geprüft werden können. Die zweite Option verringert den Aufwand bei der Erstellung der Samples. Die Transformation erfolgt direkt vor der Übersetzung der LLVM IR in den Maschinencode beziehungsweise nach Anwendung jeglicher Compileroptimierungen, sodass die Wirkung der Obfuskation auf Basis der herkömmlichen Gestalt eines Samples ohne zusätzliche Einflüsse untersucht werden kann. Es wurde der neue LLVM Pass Manager genutzt.

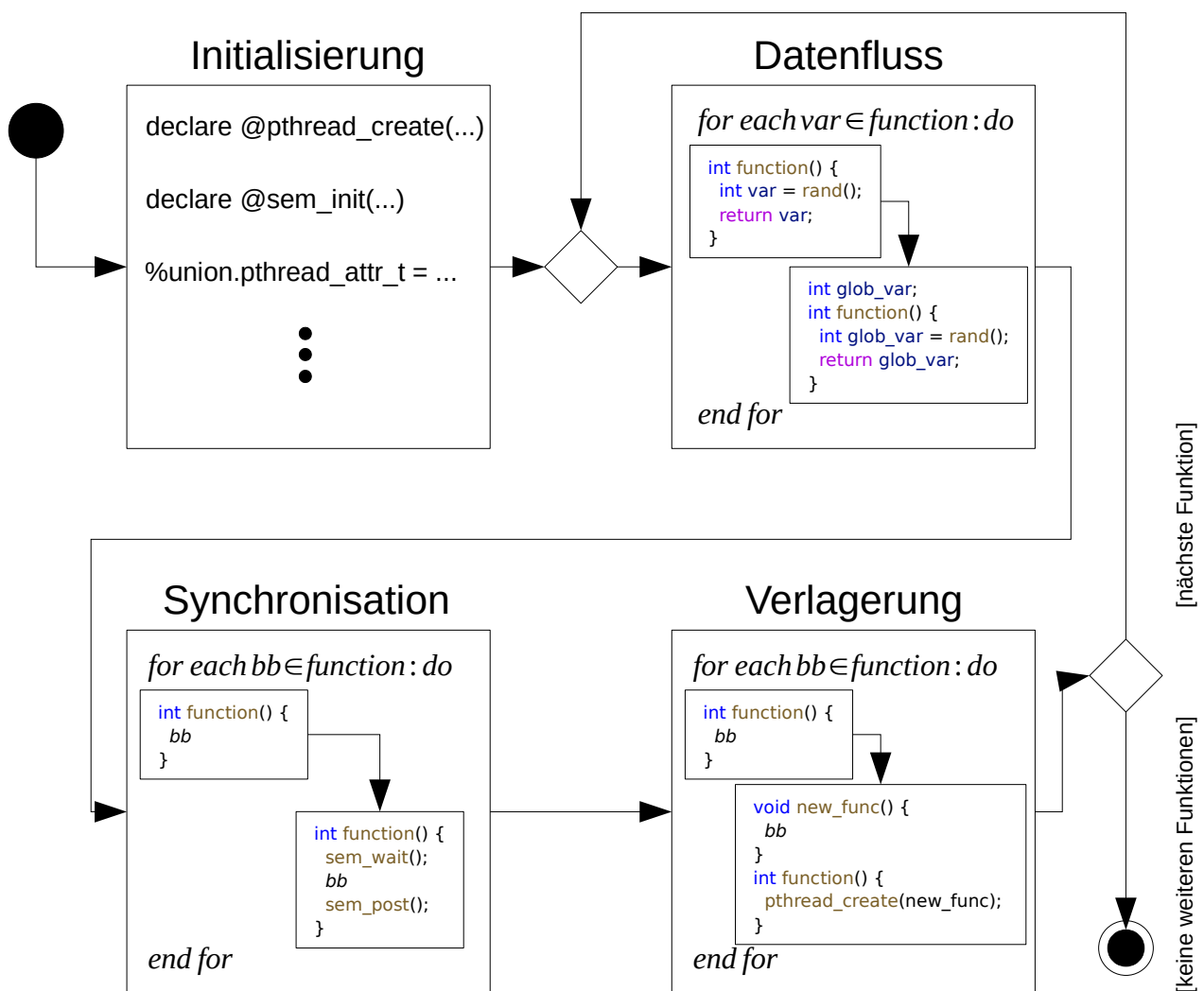


Abbildung 2: Schematische und illustrative Darstellung des Obfuskationsalgorithmus

Der Algorithmus für die Transformation weist Eigenschaften einer Pipeline Architektur auf und kann grob in vier Phasen unterteilt werden, wie in der schematischen, illustrativen Darstellung in Abbildung 2 zu erkennen ist.

In der ersten Phase erfolgt eine Initialisierung der für die Transformation genutzten Elemente; vorrangig handelt es sich hierbei um die Parallelisierungsmechanismen. Zu beachten ist, dass bei der Kompilierung beziehungsweise beim endgültigen Linken die Bibliothek anzugeben ist, welche die Parallelisierungsmechanismen bereitstellt.

Die folgenden Phasen werden für jede zu transformierende Funktion iteriert durchgeführt. Hierbei kann eine Selektion der zu inkludierenden Funktionen erfolgen, entweder durch manuelle Angabe oder durch vorherige automatische Analyse, um rechenintensive oder nicht unterstützte Funktionen auszusparen.

In der zweiten Phase findet die Verlagerung des Datenflusses in den globalen Kontext statt. Während der dritten Phase werden die Parallelisierungsmechanismen injiziert. Dieser Schritt erfolgt vor der Verlagerung der Funktionalität, da bei dieser Reihenfolge die gegenseitigen Abhängigkeiten der Basic Blocks, die während dieser letzten Phase benötigt werden, bestehen bleiben. Die dritte und letzte Phase setzt schließlich die Umgestaltung des Kontrollflusses durch das Erstellen neuer Funktionen und das Verschieben der Funktionalitäten in diese um.

3.3 Implementierung

3.3.1 Initialisierung

Abhängig vom Zielbetriebssystem müssen einige Elemente initialisiert werden. Als Elemente in diesem Sinn sind die Systemfunktionen und Systemtypen zur Realisierung der Parallelisierung der jeweiligen Plattformen zu betrachten. Die konkreten Funktionen und Typen, die für die Realisierung verwendet werden, sind in Tabelle 1 aufgelistet und dem Pendant der betrachteten Betriebssysteme gegenübergestellt.

Das *Attribut für die Erstellung von Parallelisierungsprimitiven* wird bei der Erstellung eines Threads an die jeweilige Systemfunktion übergeben und beeinflusst das Ausführungsverhalten beziehungsweise die Zugriffsrechte auf den Thread. Da für die Realisierung der Obfuskation die Standardeinstellungen bei der Thread-Erstellung ausreichend sind, wird für diesen Parameter stets ein *NULL-Pointer* übergeben.

Das *Loslösen eines Threads* ist für aufgerufene Funktionen erforderlich, die während eines Funktionsaufrufs mehrfach durchlaufen werden (wie bei Schleifen). In solchen Funktionen startet sich der Thread vor der Terminierung selbst neu. Hierbei wird der globale Thread-Identifizierer überschrieben, sodass dieser Thread nicht über die entsprechende Funktion *synchronisiert* werden kann, um die für diesen Thread allokierten Ressourcen freizugeben. Damit dies durch das Betriebssystem erfolgt, ist es erforderlich, den Thread vorher *loszulösen*. Falls die Ressourcen nicht freigegeben werden, kann es sonst, nach gewisser Ausführungsdauer und dementsprechend erheblicher Anzahl von erstellten Threads, aufgrund Limitierungen zu systembedingten Fehlern bei der Erstellung weiterer Threads kommen, was undefiniertes Verhalten zur Folge haben kann. Demgemäß erfolgt vor dem Return der

aufrufenden Funktion die *Terminierung* von noch aktiven Threads (also solchen, die nicht durchlaufen wurden oder sich selbst neu gestartet haben) und die *Synchronisation* mit diesen Threads.

Funktion bzw. Typ	Linux	Windows
Attribut für die Erstellung von Parallelisierungsprimitiven	pthread_attr_t	SECURITY_ATTRIBUTES
Erstellung eines Threads	pthread_create	CreateThread
Loslösen eines Threads	pthread_detach	CloseHandle
Terminierung eines Threads	pthread_cancel	TerminateThread
Synchronisation mit einem Thread	pthread_join	WaitForSingleObject
Semaphor	sem_t	-
Erstellung eines Semaphors	sem_init	CreateSemaphoreA
Freigeben eines Semaphors	sem_post	ReleaseSemaphore
Sperren eines Semaphors	sem_wait	WaitForSingleObject
Freigeben der Semaphorressourcen	sem_destroy	CloseHandle

Tabelle 1: Betriebssystemabhängige Funktionen und Typen für die Realisierung der Obfuskation

Semaphore werden genutzt, um die ursprüngliche, serielle Abarbeitung zu gewährleisten. Da die *Semaphore* in der Implementierung dieser Arbeit lediglich den Wert „gesperrt“ oder „frei“ annehmen, weisen sie Ähnlichkeit mit *Mutexen* auf. Der entscheidende Unterschied ist jedoch, dass bei *Mutexen* das *Sperren* und *Freigeben* durch denselben Thread erwartet wird. Versucht ein Thread ein *Mutex* freizugeben, welches nicht durch besagten Thread gesperrt wurde, tritt undefiniertes Verhalten ein [58]. Diese Freigabe aus einem anderen Thread ist jedoch bei der Obfuskation prinzipiell notwendig, da jedes *Semaphor* zu Beginn gesperrt ist und ein Thread das *Semaphor* seines jeweiligen Nachfolgers entsperrt. Als Alternative wurden ebenfalls *Spinlocks* [59] geprüft, durch die hohe Anzahl von Threads führt dies jedoch zu einem unvergleichbaren Performanzverlust, da die Ausführung des Threads nicht gestoppt wird und der Prozessor durchgehend mit der Prüfung des *Spinlocks* beschäftigt ist. Hinsichtlich des *Semaphortyps* wird bei Windows der native *HANDLE*-Typ verwendet, welcher als einfacher *void-Pointer* definiert ist, sodass hier keine gesonderte Initialisierung notwendig ist.

3.3.2 Datenfluss

In dieser Phase wird eine „Globalisierung“ des Datenflusses durchgeführt, das bedeutet lokale Identifier werden in den globalen Kontext verschoben. Da es in der LLVM IR aufgrund der Nähe zur Maschinensprache keine lokalen Variablen im herkömmlichen Sinn gibt, soll an dieser Stelle der Begriff Identifier näher erläutert werden.

Der Rückgabewert einer Instruktion wird in der LLVM IR ausnahmslos als lokaler Identifier dargestellt. Der entscheidende Unterschied ist, dass solch ein lokaler Identifier im Gegensatz zur lokalen Variablen stets einen flüchtigen Wert darstellt¹⁰. Dies entspringt dem Umstand, dass Funktions- und

¹⁰ Ausgenommen sind konstante Werte, welche in der Compilerausgabe hartkodiert werden.

Berechnungsergebnisse bei der Maschinensprache grundsätzlich in Registern vorzufinden sind. Eine lokale Variable wird bei der Kompilierung durch einen auf dem Stack allokierten Bereich realisiert, in welchem der flüchtige Wert aus dem Register abgelegt wird, sofern dieser Schritt nicht durch Compileroptimierung ausgelassen wird. Als Beispiel sind die folgenden Codeausschnitte zu betrachten.

```
1  int main(int argc, char *argv[]) {
2      int a = atoi(argv[1]);
3      int b = log(a);
4      int c = acos(b);
5      return c;
6  }
```

Codeausschnitt 3: Simpler Beispielcode in C

Basierend auf dem Beispielquellcode in Codeausschnitt 3 wurde die LLVM IR in Codeausschnitt 4 generiert (mit Compileroptimierungen der Stufe 1). Für die rechte Seite wurde der Beispielquellcode folgendermaßen modifiziert: In Codeausschnitt 3, Zeile 5 wurde `return c;` zu `return c + b;` geändert. Entsprechend beinhaltet die rechte Seite zusätzlich die Additionsinstruktion in Zeile 11, während der Code sonst identisch geblieben ist.

<pre>1 define [...] i32 @main(i32 %0, i8** [...] %1) [...] { 2 %3 = getelementptr inbounds i8*, i8** %1, i64 1 3 %4 = load i8*, i8** %3, align 8, !tbaa !2 4 %5 = call i32 @atoi(i8* %4) #3 5 %6 = sitofp i32 %5 to double 6 %7 = call double @log(double %6) #4 7 %8 = fptosi double %7 to i32 8 %9 = sitofp i32 %8 to double 9 %10 = call double @acos(double %9) #4 10 %11 = fptosi double %10 to i32 11 ret i32 %11 12 }</pre>	<pre>1 define [...] i32 @main(i32 %0, i8** [...] %1) [...] { 2 %3 = getelementptr inbounds i8*, i8** %1, i64 1 3 %4 = load i8*, i8** %3, align 8, !tbaa !2 4 %5 = call i32 @atoi(i8* %4) #3 5 %6 = sitofp i32 %5 to double 6 %7 = call double @log(double %6) #4 7 %8 = fptosi double %7 to i32 8 %9 = sitofp i32 %8 to double 9 %10 = call double @acos(double %9) #4 10 %11 = fptosi double %10 to i32 11 %12 = add nsw i32 %11, %8 12 ret i32 %12 13 }</pre>
---	--

Codeausschnitt 4: Gegenüberstellung der auf Basis von Codeausschnitt 3 generierten LLVM IR

Der nun folgende Codeausschnitt 5 stellt das Ergebnis des, auf Basis der in Codeausschnitt 4 aufgeführten einzelnen LLVM IR Ausschnitte, generierten Maschinencodes in der Assemblersprache gegenüber (die zusätzlich hinzugekommenen Zeilen wurden farblich markiert).

<pre> 1 <main>: 2 push %rax 3 mov 0x8(%rsi),%rax 4 mov %edi,0x4(%rsp) 5 mov %rax,%rdi 6 callq 4004e0 <atoi@plt> 7 cvtsi2sd %eax,%xmm0 8 callq 4004c0 <log@plt> 9 cvttsd2si %xmm0,%eax 10 cvtsi2sd %eax,%xmm0 11 callq 4004d0 <acos@plt> 12 cvttsd2si %xmm0,%eax 13 pop %rcx 14 retq </pre>	<pre> 1 <main>: 2 push %rax 3 mov 0x8(%rsi),%rax 4 mov %edi,0x4(%rsp) 5 mov %rax,%rdi 6 callq 4004e0 <atoi@plt> 7 cvtsi2sd %eax,%xmm0 8 callq 4004c0 <log@plt> 9 cvttsd2si %xmm0,%eax 10 cvtsi2sd %eax,%xmm0 11 mov %eax,(%rsp) 12 callq 4004d0 <acos@plt> 13 cvttsd2si %xmm0,%eax 14 mov (%rsp),%ecx 15 add %ecx,%eax 16 pop %rcx 17 retq </pre>
---	--

Codeausschnitt 5: Gegenüberstellung des auf Basis von Codeausschnitt 4 generierten Maschinencodes in Assemblerdarstellung (AT&T Syntax)

Wie an diesem Beispiel zu erkennen ist, sind auf der rechten Seite neben der Additionsinstruktion zwei *Move*-Instruktionen hinzugekommen, die jeweils den Rückgabewert der Funktion aus Zeile 8 (nach Konvertierung in Zeile 9) in Zeile 11 auf dem Stack ablegt und in Zeile 14 wieder vom Stack liest. Daraus folgt, dass ein Identifier lediglich den Wert darstellt, die Verwaltung dieses Werts im Hinblick auf den Speicher aber erst während der Kompilierung bestimmt wird. Diese implizit hinzugekommenen Instruktionen können in der LLVM IR auch explizit eingesetzt werden. Neben dem eigentlichen Speicher und Ladebefehl muss jedoch mit einer weiteren Instruktion der benötigte Speicherplatz auf dem Stack allokiert werden; beim Rückgabewert dieser Allokationsinstruktion handelt es sich wiederum um einen (konstanten) Identifier, der einem Zeiger auf den allokierten Stackbereich entspricht. Für die Transformation ist dies insofern relevant, als dass dieser lokale Identifier explizit durch eine separate Instruktion an einer eindeutigen, bekannten Speicheradresse abgelegt (hier realisiert durch globale Variablen) werden muss, um den Wert an einer anderen Stelle nutzen zu können.

Die Transformation gestaltet nun sich wie folgt: Zuerst wird eine globale Variable für den betrachteten lokalen Identifier erstellt und mit *null* initialisiert. Anschließend wird hinter die Instruktion, die diese lokale Variable emittiert, beziehungsweise vor den ersten Basic Block¹¹ der aktuellen Funktion (im Falle von Funktionsparametern) eine Speicherinstruktion injiziert, welche den Wert des Identifiers in der globalen Variablen speichert. Abschließend wird vor jeder Instruktion, die diesen Identifier nutzt, eine Ladeinstruktion injiziert und in der jeweiligen aktuellen Instruktion der ursprüngliche Identifier mit dem zurückgegebenen Identifier dieser Ladeinstruktion substituiert.

Hierbei findet eine Iteration über jeden Basic Block als auszulagernde Einheit statt, sodass geprüft werden kann, ob ein betrachteter Identifier in einem anderen Basic Block genutzt wird. Sollte dies

¹¹ In diesem Zusammenhang wird ein neuer Basic Block erstellt, der bei der Iteration in dieser und den nächsten Phasen nicht miteinbezogen wird

nicht der Fall sein, ist eine Auslagerung des Identifiers nicht erforderlich und wird aus Optimierungsgründen ausgelassen.

Eine besondere Behandlung erfahren PHI-Nodes, Allokationsinstruktionen und Return-Instruktionen:

PHI-Nodes stellen in der LLVM IR eine Instruktion dar, welche abhängig vom vorher durchlaufenen Basic Block einen unterschiedlichen Wert einnimmt. Ein einfaches Beispiel kann anhand einer Schleife demonstriert werden. Angenommen in dieser Schleife wird ein String mit einfacher Addition des Zeigers iteriert. Wird der Schleifenkörper erstmals durchlaufen, soll der String-Identifier aus dem vorangegangenen Basic Block für die Schleifenoperation verwendet werden. Anschließend wird die Adresse des String-Identifiers inkrementiert, wobei ein neuer Identifier emittiert wird. Wieder am Kopf der Schleife angekommen, soll nun die neue Adresse des später auftretenden Identifiers genutzt werden. Da bei der Transformation die Basic Blocks getrennt werden, müssen die PHI-Nodes durch einen anderen Mechanismus ersetzt werden. Bei diesem Vorgang erfolgen die folgenden Maßnahmen:

- Zu Beginn wird (noch vor der Behandlung der Funktionsparameter) für jede PHI-Node eine globale Variable erstellt und die jeweilige PHI-Node durch eine Instruktion ersetzt, welche den Wert der globalen Variablen lädt.
- Hinter die Instruktionen, die den einzelnen Identifier emittieren, die durch die PHI-Node verarbeitet werden, beziehungsweise an das Ende eines zu dieser PHI-Node abhängigen Basic Blocks (im Falle von Konstanten), wird die korrespondierende Speicherinstruktion für die globale Variable eingefügt.
- Des Weiteren müssen zwei Spezialfälle berücksichtigt werden:
 - Der erste entsteht, wenn eine PHI-Node abhängig vom Basic Block ist, in welcher sich diese PHI-Node befindet, und diese PHI-Node auch in einem anderen Basic Block genutzt wird. Da der Wert der PHI-Node in der globalen Variablen am Ende des Basic Blocks stets überschrieben wird, ist der ursprüngliche Wert für die Nutzung in einem anderen Basic Block nicht mehr vorhanden. Deshalb wird der aktuelle Wert der globalen Variablen einer PHI-Node an der Stelle der ursprünglichen PHI-Node in eine weitere globale Variable geladen, welche im weiteren Verlauf des Programms genutzt wird.
 - Der zweite Spezialfall ergibt sich bei einer Verkettung von PHI-Nodes, also wenn eine PHI-Node den Wert einer anderen PHI-Node nutzt. Da die PHI-Nodes vollständig entfernt werden und somit die Verbindung zwischen zwei derartigen Instruktionen verloren geht, wird an der Stelle der benutzten PHI-Node der Wert der globalen Variablen in die nutzende PHI-Node übertragen.

Da der emittierte Identifier einer *Allokationsinstruktion* immer einem Zeiger auf einen Speicherbereich entspricht, ist die Injektion von Lade- und Speicherinstruktionen unnötig, sodass dieser lokale Identifier direkt durch einen globalen Identifier (welcher auch immer ein Zeiger ist) ersetzt werden kann.

Sollte die letzte Instruktion eines Basic Blocks eine *Return-Instruktion* mit einem Rückgabewert sein, so wird der Rückgabewert in einer für die ursprüngliche Funktion angelegten globalen Variable gespeichert und durch den Wert *void* ersetzt¹².

3.3.3 Synchronisation

Die Synchronisation der in der nächsten Phase programmatisch erstellten Threads zur Einhaltung der ursprünglichen seriellen Abarbeitungsreihenfolge wird umgesetzt, indem für jeden Basic Block (mit Ausnahme des ersten Basic Blocks der aktuellen Funktion) ein globales Semaphor am Anfang der ursprünglichen Funktion initiiert wird und an den Anfang jedes Basic Blocks ein Wartepunkt durch die Funktion zum *Sperren eines Semaphors* eingefügt wird.

Das Semaphor soll je nach Ergebnis der Terminator-Instruktion eines der vorangegangenen Basic Blocks freigegeben werden. Bei einer Terminator-Instruktion handelt es sich um die letzte Instruktion eines Basic Blocks, welche den weiteren Programmablauf steuert. Der gängigste Terminator ist die Verzweigungsinstruktion, auch Return-Instruktionen zählen hierzu.

Um das *Freigeben eines Semaphors* zu realisieren, wird schließlich bei der Terminator-Instruktion (ausgenommen Return-Instruktion, siehe unten) jedes Vorgängers des aktuellen Basic Blocks für die Bedingungen, die den aktuellen Basic Block als Nachfolger bestimmen, ein neu erstellter Basic Block eingesetzt, in welchem die entsprechende Systemfunktion aufgerufen wird.

Handelt es sich bei der letzten Instruktion eines Basic Blocks um eine Return-Instruktion (die zwangsläufig in jeder Funktion existiert), wird unmittelbar vor dieser Instruktion der Aufruf zum *Freigeben eines Semaphors* eingefügt. Dieser Fall wird gesondert betrachtet, da ein Basic Block mit einer Return-Instruktion als Terminator keinen Nachfolger hat und somit kein Vorgänger sein kann. Der diesem Semaphor zugehörige Wartepunkt wird während der nächsten Phase in der ursprünglichen Funktion nach der letzten Funktion zur *Erstellung eines Threads* eingesetzt und signalisiert somit der aufrufenden Funktion, dass die Aufgabe der Funktion abgearbeitet wurde und die Aufräumarbeiten mit abschließendem Return beginnen können.

3.3.4 Verlagerung

In dieser Phase werden schlussendlich die einzelnen Basic Blocks aus der aktuellen Funktion in eine neue Funktion verschoben und in der aufrufenden Funktion die notwendigen Systemaufrufe eingefügt.

Zunächst wird eine neue Funktion ohne Parameter und Rückgabewert erstellt. Anschließend werden der aktuelle Basic Block und (falls vorhanden) die Nachfolger gemäß der Terminator-Instruktion in diese Funktion verschoben. Hierbei ist zu beachten, dass es sich (wie in der vorangegangenen Phase beschrieben) bei den Nachfolgern um die Basic Blocks handelt, welche das Semaphor des ursprünglichen Nachfolgers freigeben. Außerdem wird während dieses Vorgangs für jede neue Funktion genau eine globale Variable für den Thread-Identifizier erstellt. Zum einen ist dies nötig, um nach Abarbeitung der Funktion die allokierten Ressourcen der Threads freizugeben. Zum anderen soll ein Thread, der sich selbst neu startet, den neuen Identifizier so ablegen, dass die ursprüngliche Funktion während der Aufräumarbeiten darauf zugreifen kann (siehe unten). In der aufrufenden Funktion

¹² Alternativ hätte der Rückgabewert über eine Systemfunktion extrahiert werden können. Aufgrund des zusätzlichen Verwaltungsaufwand bei dieser Alternative ist der Datenfluss über eine globale Variable vorzuziehen.

wird für jede so neu erstellte Funktion ein Systemaufruf für die *Erstellung eines Threads* kreiert und der Funktionszeiger der neu erstellten Funktion als Argument übergeben.

Falls festgestellt wird, dass der aktuelle Basic Block mehrfach durchlaufen wird, findet an dieser Stelle noch die Injektion des Systemaufrufs zum *Loslösen des Threads* und zur *Erstellung eines Threads* vor der Return-Instruktion der neuen Funktion statt. Hiermit wird, wie bereits im Unterabschnitt „Initialisierung“ erörtert, das ressourcenschonende Neustarten des Threads realisiert. Die Menge der betroffenen Basic Blocks wird erfasst, indem zu einem aktuell betrachteten Basic Block rekursiv geprüft wird, ob einer der Vorgänger der aktuelle Basic Block selbst ist. Um den Prozess bei einer Schleife eines Vorgängers, ohne dass der aktuelle Basic Block sich in der Schleife befindet, terminieren zu lassen und zur Laufzeitoptimierung des Transformationsprozesses fließen bei der Untersuchung die bereits besuchten beziehungsweise die Ergebnisse zu den bereits untersuchten Basic Blocks mit ein.

Nachdem jeder Basic Block der aktuellen Funktion behandelt wurde, wird der im Unterabschnitt „Synchronisation“ bereits erwähnte Wartepunkt, dessen Freigabe nach Erfüllung der eigentlichen Aufgabe der Funktion erfolgt, in der aufrufenden Funktion eingefügt. Im Anschluss werden die Ressourcen freigegeben, indem jeder Thread-Identifizierer beziehungsweise jedes Semaphor als Argument an den Systemaufruf *Synchronisation mit einem Thread* beziehungsweise *Freigeben der Semaphorressourcen* übergeben wird.

Abschließend wird noch der Rückgabewert aus der hierfür erstellen globalen Variable geladen (siehe Unterabschnitt „Datenfluss“) und per Return-Instruktion zurückgegeben.

Um auszuschließen, dass die Reihenfolge, in der die Threads erstellt werden, einen Einfluss auf das Ausführungsverhalten des Programms hat, erfolgt in der Implementierung für diese Arbeit die Erstellung der Threads in der umgekehrten Reihenfolge bezogen auf die ursprüngliche Anordnung der Basic Blocks. Die Effektivität der Obfuskation sowie die polymorphen Eigenschaften der Transformation kann durch eine zufällige Reihenfolge gesteigert werden.

3.3.5 Besonderheit beim Windows Betriebssystem

Bei der Übertragung der auf Linux funktionierenden Entwicklungsergebnisse musste auf dem Windows Betriebssystem festgestellt werden, dass sich bei steigender Ausführungsdauer die Wahrscheinlichkeit erhöht, dass der Prozess in einen Deadlock gerät. Dieses Problem ist auf die Funktion zur *Terminierung eines Threads* zurückzuführen. Wohingegen auf dem Linux OS einem Thread lediglich eine Terminierungsanfrage signalisiert wird, sodass sich der Thread kontrolliert beenden kann [60], erfolgt die Beendigung des Threads beim Windows OS unverzüglich [61]. Hierdurch entsteht die Möglichkeit, dass Threads während der Initialisierung oder Beendigung unterbrochen werden, sodass von den Threads akquirierte kritische Sektionen (ähnlich Mutexen) nicht freigegeben werden. Nachfolgend gestartete Threads warten auf die Freigabe dieser Synchronisationsmechanismen, welche jedoch aufgrund der beschriebenen Umstände niemals erfolgt.

Verschiedene Möglichkeiten bieten sich an, um dieses Problem zu lösen. Ansätze, bei denen die Priorität des Threads, welcher die Terminierungen durchführt, herabgesetzt wurde und die Ausführung besagten Threads durch Funktionsaufrufe wie *SwitchToThread* oder *Sleep(0)* verzögert wurde, blieben erfolglos. Um zusätzlich notwendige Maßnahmen bei der Transformation und beim obfuski-

erten Programm zu minimieren, könnten undokumentierte Systemfunktionen wie *NtQueryInformationThread* oder *NtQuerySemaphore* genutzt werden, um Aussagen über den Zustand des Threads treffen und die Terminierung falls erforderlich verzögern zu können. Auch wäre es möglich, die betroffenen kritischen Sektionen während der Aufrufe der Funktion zur *Terminierung eines Threads* selbst zu akquirieren. Beide Lösungen sind jedoch aufgrund der Nutzung inoffizieller Methoden und der damit verbundenen Unsicherheiten nur eingeschränkt zu bevorzugen. Ein offizieller Ansatz würde das zusätzliche einbringen von Synchronisierungsmechanismen vorsehen.

Für die vorliegende Implementierung wurde eine Lösung gewählt, bei der eine globale Zählervariable vor der Erstellung jedes Threads inkrementiert wird und durch die gestarteten Threads vor dem Erreichen des Wartepunkts dekrementiert wird. Während des Lese- und Schreibvorgangs werden *Slim Reader/Writer Locks* eingesetzt, welche mit für Geschwindigkeit optimierten Mutexen zu vergleichen sind [62], sodass durch den exklusiven Zugriff auf die Variable keine Fehler im Zusammenhang mit der Parallelisierung eintreten. Mit Hilfe dieser Variablen wird vor der Terminierung der Threads geprüft, ob sich Threads in der Initialisierung befinden und demnach potentiell Synchronisierungsprimitiven akquiriert haben. Um die Terminierung eines Threads während der Beendigung des Threads zu unterbinden, erfolgt darüber hinaus der Aufruf zur *Terminierung eines Threads* durch die jeweiligen Threads anstelle der Return-Instruktion.

Diese Lösung wird präferiert, da sie am wenigsten Logik zur komplexen Transformation hinzufügt. Außerdem ist der hinzukommende Verwaltungsaufwand beim obfuskierten Programm gering, wodurch die Performanz des Programms nicht übermäßig beeinflusst wird. Die Korrektheit dieser Lösung wird im Unterabschnitt „Vergleich zwischen obfuskiertem und nicht obfuskiertem Programm“ verifiziert.

4 Evaluation

In diesem Kapitel erfolgt die Evaluation der vorgestellten Obfuskationstechnik. Hierfür sollen an dieser Stelle die in den nachfolgenden Abschnitten genutzten Metriken und Untersuchungsmethoden vorgestellt und in einen Kontext gesetzt werden.

Nach der Verifikation, dass bei den obfuskierten Samples von einer entsprechenden Funktionsfähigkeit auszugehen ist, sollen formelle Obfuskationsmetriken gemäß [2] zur Bewertung der vorgestellten Implementierung herangezogen werden. Hierbei wird die Qualität dreidimensional anhand der Wirksamkeit, der Widerstandsfähigkeit und der Kosten bestimmt. In den anschließenden Abschnitten sollen die Auswirkungen des Obfuskationsprozesses informell, aber praxisnah untersucht werden. Hierbei wird jeweils ein Bezug zum Reverse Engineering und zur Malware Detektion hergestellt. Zu erwarten ist, dass sich bei der statischen Verhaltensanalyse ohne Kenntnis dieser Methode kein aussagekräftiger Kontrollflussgraph erstellen lässt. Hinsichtlich der Malware Detektion ist eine geringere Detektionsrate als Ziel definiert.

Um die Evaluation möglichst praxisnah zu gestalten, wurden zum einen bekannte open-source Softwaretools dem Transformationsprozess unterzogen. Zum anderen soll auch eine Einschätzung bezüglich den Auswirkungen auf die Malware Detektion vorgenommen werden, sodass Malware Samples erforderlich sind. Die Gesamtheit der genutzten Virussamples bildet sich aus allen open-source Malware Projekten, die während der Arbeit an diesem Projekt aufgefunden werden konnten, deren Kompilierung mit akzeptablem Aufwand möglich war und bei denen das Kompilat durch mindestens einen Scanner auf VirusTotal detektiert wurde.

Da im Gegensatz zu Virussamples als nativ ausführbare Datei die Anzahl an Quellcodes für Malware begrenzt ist, wurde in diesem Zusammenhang ebenfalls die Möglichkeit untersucht ausführbare Dateien in die LLVM IR zu überführen. Dieser Ansatz stellt eine Binary Rewriting Technik unter vollständiger Übersetzung der ausführbaren Datei dar [10]. Auf Anwendbarkeit wurde das nach eigenen Angaben elaborierteste Tool McSema [63] und die dort aufgeführten Alternativen geprüft. Bei McSema wird jedoch für optimale Kompatibilität das kommerzielle IDA Pro Reverse Engineering Tool [64] benötigt, welches bei Erstellung dieser Arbeit nicht zur Verfügung stand. Die meisten der weiteren Alternativen sind nicht mehr aktuell und/oder unzuverlässig. So war bei der Entwicklung der vielversprechendsten Alternative llvm-mctoll nie das Ziel, eine wieder kompilierbare LLVM IR zu generieren [65]. Dementsprechend müssen Binary Rewriting Methoden (wie bereits im Abschnitt „Obfuskation“ angedeutet) für die Evaluation ausgeschlossen werden.

Im Folgenden erfolgt die Auflistung der für die Evaluation verwendeten open-source Software:

- **BBP Formel:** Ein auf Github zur Verfügung gestellter Code zur Berechnung der Kreiszahl Pi mit der Bailey-Borwein-Plouffe-Formel in der Programmiersprache C++. [66]
- **bzip2:** Ein Programm und eine Bibliothek für verlustfreie Kompression in der Programmiersprache C (Version 1.0.8). [67]
- **Coreutils:** Standardprogramme des GNU Betriebssystems [68]. Ein für die Evaluation beliebtes Softwarepaket, welches auch in der referenzierten Literatur genutzt wurde (nur Linux, Version 8.28) [46, 69].

- **IoT Malware:** Ein Repository mit Quellcode für Malware, die insbesondere auf Geräten eingesetzt werden, die dem Internet of Things beigemessen werden (nur Linux). [70]
 - **BASHLITE:** Ein Remote Access Tool basierend auf dem Client-Server-Prinzip.
 - **Lightaidra:** Ein Remote Access Tool basierend auf Kommunikation über Internet Relay Chat, insbesondere für Router.
 - **Mirai IoT Botnet:** Ein Computerwurm, mit welchem Bot-Netze aufgebaut werden können.
 - **pnsnscan:** Ein Netzwerkscanner, der Parallelität nutzt.
- **Mettle:** Eine native Implementierung des Meterpreters mit hoher Portabilität (nur Linux, Version 1.0.9). [71]
- **VX Heaven:** Eine Virendatenbank zu Forschungszwecken, welche auch von anderen wissenschaftlichen Publikationen herangezogen wurde [33, 34, 40, 69, 72, 73, 74]. Die offizielle Website befand sich im Fokus polizeilicher Ermittlungen [75] und war während der Erstellung dieser Arbeit nicht erreichbar. Die Datenbank in der Version vom 18. Mai 2010 konnte dennoch von einer anderen Quelle bezogen werden [76]. In dieser circa 48 GB großen Datenbank befinden sich hauptsächlich Dateien im Binärformat. Für die folgenden Untersuchungen wurden jedoch auch fünf Dateien mit kompilierbarem C-Quellcode gefunden, welche im folgenden aufgelistet sind (nur Linux). Da die Datenbank keine näheren Informationen enthält, kann keine Beschreibung der Funktion der einzelnen Programme erfolgen.
 - Exploit.Linux.Interbase, Exploit.Linux.Nhttpd, Net-Worm.Linux.Slapper, Virus.Unix.A-drastea, Worm.FreeBSD.Block
- **TheZoo:** Ein Repository mit Malware Samples, in welchem auch Quellcodes zur Verfügung gestellt werden (nur Windows). [48]
 - Alina, AryanRAT, Carberp, CyberBot, Dexter, Dokan, Grum, LiquidBot, MyDoom.A, ogw0rm, rBot, X0R-USB, xTBot
- **Meterpreter:** Die bekannteste Payload des Metasploit Frameworks (nur Windows, Version 2.0.45). [77]
- **Mimikatz:** Ein Tool zur Extraktion von sensiblen Nutzerdaten auf einem Windows Betriebssystem (nur Windows, Version 2.2.0-20210709). [78]
- **Polymorphic Virus:** Eine Implementierung einer polymorphen Malware für Testzwecke (nur Windows). [79]
- **ReflectiveDllInjection:** Eine Technik zur Injektion eines Programms in einen anderen Prozess mit Hilfe von reflexiver Programmierung (nur Windows). [80]

Die Linux-Samples wurden auf einem Linux Mint 19.1 (4.15.0-147-generic, 64-bit) mit GCC Version 7.5.0 beziehungsweise Clang Version 10.0.0 kompiliert. Die Windows-Samples wurden auf einem Windows 10 (21H1, 19043.1052, 64-bit) mit MSVC Version 19.29.30038.1 beziehungsweise Clang Version 11.0.0 kompiliert. Bei der Kompilierung wurde die Optimierungsstufe 2 angewandt. Ausgenommen hiervon sind 32-bit Windows-Samples, weil die mit Clang kompilierten Programme

unabhängig von der Obfuskation bei Anwendung jeglicher Optimierung nicht funktionsfähig waren. Für die Ausführung des Programms unnötige Symbole wurden bei den Linux Samples entfernt. Bei einigen Windows Malware Samples war die Kompilierung nicht möglich, weil das `__asm`-Schlüsselwort benutzt wurde und dies bei der 64-bit Kompilierung nicht unterstützt wird. Bei einigen älteren Projekten wurden minimale Anpassungen vorgenommen, um die Kompilierfähigkeit herzustellen. Bei Mimikatz wurden die Ausnahmebehandlungen aus dem Quellcode entfernt.

4.1 Beweis der Korrektheit des obfuskierten Programms

4.1.1 Vergleich zwischen obfuskiertem und nicht obfuskiertem Programm

Die Beschreibung der Implementierung indiziert bereits, dass die Semantik des transformierten Programms erhalten bleibt und deshalb bei gleichen Programmeingaben identische Programmausgaben zu erwarten sind. Aufgrund der Eventualität, dass unbekannte Sprachkonstrukte in der LLVM IR zu falschen Annahmen bei der Transformation und schließlich zu nicht funktionsfähigen Erzeugnissen führen können, soll an dieser Stelle ein komplexeres obfuskiertes Programm einem Stresstest unterzogen werden. Insofern das beobachtbare Verhalten der ausführbaren Datei identisch mit einer nicht obfuskierten Referenzdatei ist, kann von einer hinreichenden Wahrscheinlichkeit hinsichtlich der Korrektheit von anderen ähnlich komplexen Programmen ausgegangen werden.

Für diesen Ansatz wird das Programm `bzip2` für verlustfreie Kompression von Dateien als geeignet befunden. Das Programm umfasst 4383 Codezeilen und das ELF-Kompilat weist eine Größe von circa 100 KB auf, das obfuskierte Programm ist knapp 650 KB groß. Beim Windows OS weist das nicht obfuskierte PE-Kompilat eine Größe von etwa 240 KB auf und die obfuskierte Datei ist ungefähr 1,1 MB beziehungsweise 2,3 MB bei der 32-bit Variante groß. Es besteht die Möglichkeit verschiedene Argumente zu übergeben und als Ausführungsergebnis lassen sich Konsolenausgaben sowie durchgeführte Dateiumwandlungen der Kompression vergleichen.

Die Kommandozeileingaben in Codeausschnitt 6 wurden jeweils auf das obfuskierte und das nicht obfuskierte Programm, 32-bit und 64-bit, Linux OS und Windows OS angewandt¹³. Die betroffenen Referenzdateien, deren Inhalt zufällig generiert wurde und deren Name die Dateigröße angibt, wurden nach jedem Befehl mithilfe des `diff` Kommandos [81] miteinander verglichen. Sofern Konsolenausgaben erfolgten, wurden diese einer Sichtprüfung unterzogen. Es konnten keine Unterschiede zwischen obfuskiertem und nicht obfuskiertem Programm festgestellt werden. Auch bei einem Keyboard Interrupt während längerer Ausführungsphasen verhielt sich das Programm in jeglichen vorliegenden Formen identisch.

Hinsichtlich der Besonderheit bei Windows gemäß dem vorangegangenen Unterabschnitt „Besonderheit beim Windows Betriebssystem“ wurde bei den beschriebenen Versuchen kein Deadlock festgestellt. Die längste Ausführungsdauer des Programms unter Kompilierung der 1 MB großen Datei umfasste mehrere Stunden¹⁴. Ohne die Maßnahmen im genannten Unterabschnitt trat der Deadlock bereits nach wenigen Sekunden bis Minuten ein, sodass die Funktionsfähigkeit der obfuskierten Programme für das vorliegende Einsatzszenario als gegeben angenommen werden kann.

¹³ Auf dem Linux OS wurde für die 32-bit Version die Standardgröße des Stackspeichers auf 1 MiB herabgesetzt.

¹⁴ Aussagen zur Performanz werden im weiteren Verlauf des Kapitels getätigt.


```

$ ./bzip2
$ ./bzip2 --help
$ ./bzip2 --best tst10K
$ ./bzip2 -d tst10K.bz2
$ ./bzip2 -k -v tst100K
$ ./bzip2 -L -V
$ ./bzip2 -s -1 tst1M
$ ./bzip2 -d -9 tst1M.bz2
$ ./bzip2 -g
$ ./bzip2 -c tst10B > out.txt

```

Codeausschnitt 6: Kommandozeilenbefehle zur Prüfung der Korrektheit des obfuskierten Programms

4.1.2 Coreutils Testsuite

Neben dem manuellen Vergleich zwischen einem obfuskierten und nicht obfuskierten Kompilat bieten diverse Softwarepakete automatische Tests an. Um einen Eindruck von der Zuverlässigkeit der vorliegenden Implementierung zu ermöglichen, sollen an dieser Stelle die Testergebnisse des Softwarepakets Coreutils dargestellt und betrachtet werden. Das Softwarepaket Coreutils bietet zwei eigenständige Testsuiten, wovon eine unabhängig vom eigentlichen Softwarepaket Coreutils entwickelt wird, nämlich die Gnulib Tests [82]. Wohingegen die Coreutils Tests skriptbasiert sind und die kompilierten Binärdateien unter Beobachtung ausführen, handelt es sich bei den Gnulib Tests um C-Programme, die gegen die kompilierte statische Coreutils-Bibliothek gelinkt werden und deren Ausgabe den Ausgang der Tests indiziert.

Die Tests wurden in den Standardeinstellungen mit 64-bit Kompilaten¹⁵ durchgeführt. Da vereinzelt die Funktion *signal* im Softwarepaket verwendet wird, diese Funktion jedoch bei Multithread-Prozessen zu Problemen führen kann [83], wurde auf eine Obfuskation von Modulen, die diese Funktion nutzen, verzichtet¹⁶. Obwohl dies die Anzahl der Deadlocks während der Durchführung der Tests erheblich reduziert hat, funktionierte das Coreutils-Programm *timeout* nicht absolut zuverlässig. Dies ist dadurch zu begründen, dass die Parallelität der Obfuskation trotzdem in anderen Modulen, welche gemeinsam mit dem *timeout*-Modul das Programm bilden, vorhanden war. Da das Programm jedoch in etlichen Skripten eingesetzt wurde, um einem zu testenden Programm nur eine beschränkte Zeitspanne für die erfolgreiche Ausführung zur Verfügung zu stellen, wurde das komplette *timeout*-Programm nicht obfuskiert. Darüber hinaus mussten neun¹⁷ Coreutils Tests und zwei¹⁸ Gnulib Tests aus verschiedenen Gründen deaktiviert werden. Teilweise traten Deadlocks aus ähnlichen Gründen wie beim *timeout*-Programm auf. Überwiegend jedoch wurden in einigen Testskripten Aktionen durchgeführt, welche auf ein obfuskiertes Programm eine andere Wirkung entfalten, insbesondere das Setzen eines Speicherlimits um beispielsweise einen Speicherfehler herbeizuführen. Statt dem Auftreten des Speicherfehlers schlug die Erstellung der Threads fehl, wodurch die funktionale Abarbeitung nicht stattfinden konnte und die aufrufende Funktion am letzten Wartepunkt hing.

¹⁵ Die Testergebnisse mit 32-bit Kompilaten sind vergleichbar, weshalb auf diese nicht gesondert eingegangen wird.

¹⁶ Dies trifft nur auf die Tests zu, bei den folgenden Abschnitten wurde die komplette Obfuskation angewandt.

¹⁷ *xstrtol*, *cut-huge-range*, *csplit-heap*, *factor-parallel*, *head-c*, *printf-surprise*, *sort-discrim*, *line-bytes*, *tr*

¹⁸ *test-hash*, *test-uc_width2*

obfuskiert

Coreutils	Gnulib
=====	=====
# TOTAL: 591	# TOTAL: 320
# PASS: 353	# PASS: 200
# SKIP: 128	# SKIP: 37
# XFAIL: 0	# XFAIL: 0
# FAIL: 89	# FAIL: 83
# XPASS: 0	# XPASS: 0
# ERROR: 21	# ERROR: 0

nicht obfuskiert

Coreutils	Gnulib
=====	=====
# TOTAL: 600	# TOTAL: 322
# PASS: 485	# PASS: 286
# SKIP: 109	# SKIP: 36
# XFAIL: 0	# XFAIL: 0
# FAIL: 0	# FAIL: 0
# XPASS: 0	# XPASS: 0
# ERROR: 6	# ERROR: 0

Codeausschnitt 7: Testergebnisse des Coreutils Softwarepakets

In Codeausschnitt 7 sind die Testergebnisse gegenübergestellt. Auf der linken Seite sind die Ergebnisse des obfuskierten und auf der rechten Seite die des nicht obfuskierten Softwarepakets dargestellt. Die meisten ausgelassenen Tests sind dadurch zu begründen, dass sie entweder zu aufwendig sind oder als Superuser ausgeführt werden müssen. Auf diese zusätzlichen Tests wurde verzichtet, weil die Anzahl der Standardtests als ausreichend befunden wurde. Ferner konnten für einige Tests die Voraussetzungen nicht erfüllt werden, die insbesondere auf die Systemumgebung, aber auch auf die Funktionsfähigkeit von bestimmten Konstrukten wie der *signal*-Funktion zurückzuführen sind, weshalb die Anzahl der ausgelassenen Tests bei der obfuskierten Variante höher ist. Während die Erfolgsquote beim nicht obfuskierten Softwarepaket erwartungsgemäß bei nahezu 100 % liegt, hat die obfuskierte Variante fast 75 % aller durchgeführten Tests erfolgreich absolviert. Bei den Gnulib Tests ist zu beachten, dass auch die Testprogramme obfuskiert wurden. Bei nicht obfuskierten Testprogrammen reduziert sich die Anzahl der fehlgeschlagenen Tests auf 41, wodurch die Gesamterfolgsquote auf circa 80 % steigt. Wird hierbei der Umstand bedacht, dass einige fehlgeschlagenen Tests auf der Obfuskation inhärente Problematiken zurückzuführen sind, kann die Stabilität der Obfuskation für die vorgesehenen Einsatzszenarien als genügend eingeordnet werden.

4.2 Formelle Obfuskationsmetriken

Die in diesem Abschnitt verwendeten Metriken sind hauptsächlich aus [2] entnommen, weshalb im Weiteren auf die Angabe dieser Referenz verzichtet wird. Des Weiteren sollen nur die Ergebnisse vorgestellt werden, die Messdaten sind dem Anhang zu entnehmen. Um die Komplexität überschaubar zu halten, erfolgt bei den praktischen Messungen – sofern nicht anders angegeben – eine Beschränkung auf die Programme des Coreutils Softwarepakets. Dieses repräsentative Softwarepaket bietet 105 Programme mit einer Größe von circa 20 bis 130 KB und erfüllt somit die Eigenschaften der Zielgruppe, denn eine vollumfängliche Obfuskation von Nutzerapplikationen abseits der Konsolentypischen Handhabung wird nicht empfohlen. Sofern sich die Ergebnisse der 64-bit und 32-bit Varianten ähneln, werden nur die Ergebnisse der 64-bit Variante vorgestellt.

4.2.1 Wirksamkeit

„[...] the potency of a transformation measures how much more difficult the obfuscated code is to understand (for a human) than the original code.“ [2]

Auch wenn die Wirksamkeit insbesondere beim Reverse Engineering abhängig von der individuellen Kunstfertigkeit des *Ingenieurs* ist, soll an dieser Stelle eine Bewertung der Wirksamkeit der

Transformation anhand formeller Metriken erfolgen. Die folgenden Software Komplexitätsmetriken basieren auf der Anzahl textueller Eigenschaften von Quellcode und entstammen der Software Engineering Disziplin, wobei das Qualitätsmaß im Kontext der Obfuskation dem des Software Engineering entgegengerichtet ist. Zudem ist zu beachten, dass eine Zunahme an Informationen im Zusammenhang mit der Wirksamkeit als für die Obfuskation förderlich betrachtet wird, wohingegen der hierdurch zusätzlich benötigte Speicherplatz und die erhöhte Ausführungsdauer bezüglich der Kosten einen negativen Einfluss hat.

Die Zunahme ist bei der Thread-basierten Obfuskation abhängig von der Anzahl neu erstellter Funktionen und somit von der Einteilungsgröße. Für die praktischen Messungen wurde die in der Implementierung gewählte Einteilungsgröße von einem Basic Block verwendet.

Zunahme der Programmgröße¹⁹

Neben der Verlagerung von Funktionalität, wodurch die Programmgröße nicht verändert wird, werden neue Funktionen und Instruktionen erstellt, welche maßgeblich die Zunahme des Programms beeinflussen. Zur Bestimmung dieser Metrik sollen praktische Messungen durchgeführt werden, da eine rechnerische Bestimmung aufgrund variierender Längen hinzugefügter Instruktionen und eine detaillierte Bestimmung der Größe jeder Zusatzinformation einen unverhältnismäßigen Aufwand impliziert. Obwohl die Obfuskation vorrangig den Maschinencode betrifft (und keine konstanten Daten oder Ressourcen wie Zeichenketten oder Symbole), erstreckt sich die Zunahme auf weitere Sektionen neben der eigentlichen Codesektion. Beispielsweise nimmt die Größe der *.eh_frame-section* in ELF-Dateien zu, weil in dieser Sektion für jede Funktion in der Codesektion Informationen für die Ausnahmebehandlung bereitgestellt werden und sich die Anzahl der Funktionen erhöht. Aus diesem Grund bezieht sich die Zunahme der Programmgröße auf die Größe der gesamten Binärdatei.

In Abbildung 3 ist die Abhängigkeit zwischen ursprünglicher Programmgröße und der prozentualen Zunahme durch die Obfuskation aufgezeigt. Die geringste Zunahme beträgt 193,53 % und die höchste Zunahme liegt bei 1460,49 %. Trotz dieser hohen Distanz lässt sich auch bei Programmen mit höherer ursprünglicher Größe ein Trend zum arithmetischen Mittelwert von 386,33 % erkennen; die Mittelwertlinie ist in der Abbildung eingezeichnet. So liegen auch knapp 75 % der prozentualen Zunahmen unter dem genannten Mittelwert.

Die Zunahme der Programmgröße ist durch die hinzukommenden Instruktionen zur Initialisierung der Bestandteile der Obfuskation (also Threads, Semaphore, *et cetera*) und die Synchronisationsmechanismen besonders abhängig von der Einteilungsgröße. Hinzu kommt indes durch die Verlagerung des Datenflusses auch die Abhängigkeit von der Anzahl der Variablen, die in verschiedenen Einteilungsblöcken genutzt werden, weil hierbei zusätzliche Lade- und Speicherinstruktionen notwendig werden. Hierdurch lässt sich erklären, dass bei den vorliegenden Ergebnissen einige Programme mit wenigen Instruktionen pro Basic Block und einer hohen Anzahl von lokalen Variablen, die in mehreren Basic Blocks genutzt werden, Spitzenwerte über 600 % erreichen. Trotzdem kann bezüglich der Programmgröße von einer konstanten Zunahme der Programmgröße ausgegangen werden, weil bei höherer ursprünglicher Programmgröße eine Kompensation der Dichte dieser speziellen Eigenschaften wahrscheinlicher wird.

¹⁹ siehe Anhang 7.1 Messdaten zur Zunahme der Programmgröße

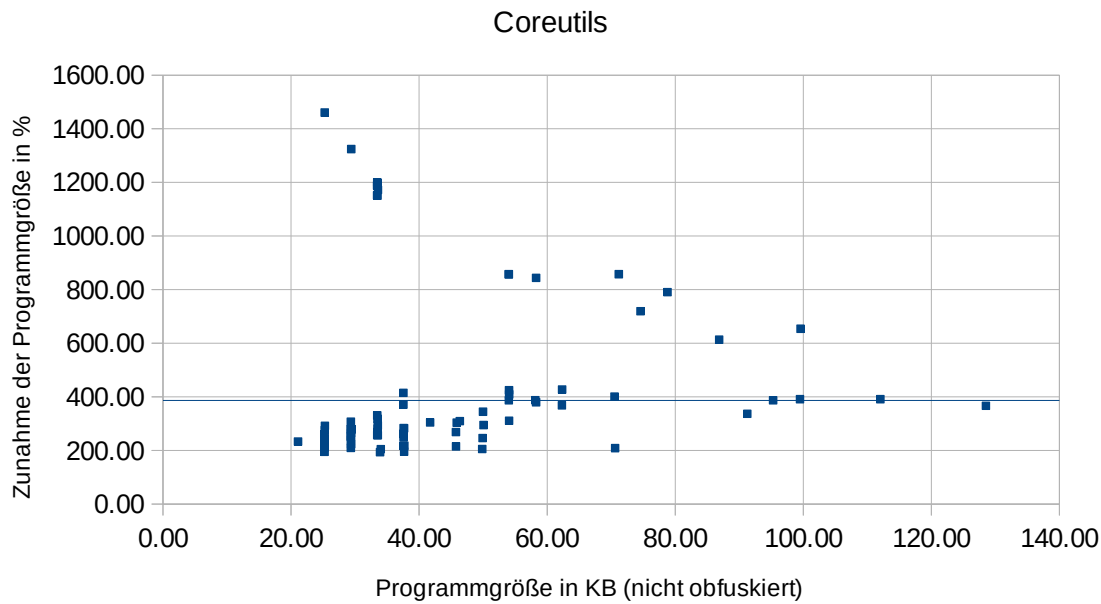


Abbildung 3: Zunahme der Programmgröße

Qualitativ lässt sich angeben, dass ein obfuskiertes Programm etwa das vier- bis fünffache der Größe eines nicht obfuskierten Programms einnimmt. Dieses Ergebnis ist jedoch erst im Zusammenhang mit der Widerstandsfähigkeit und den Kosten aussagekräftig. Zum einen, weil die Programmgröße theoretisch unlimitiert erhöhbar ist, jedoch ohne die Widerstandsfähigkeit zu erhöhen. Zum anderen, weil ein zu großes Programm zu auffällig hinsichtlich der Malware Evasion sein kann und weil der Festplattenspeicher begrenzt ist. Letzteres kann hingegen mit den heutigen Festplattenkapazitäten und bei den hier betrachteten Maßstäben als weniger schwerwiegend betrachtet werden.

Zunahme von Klassen und Methoden

Hinsichtlich objektorientierter Komponenten erfolgt keine Veränderung. Dennoch soll an dieser Stelle der Begriff „Methode“ losgelöst vom objektorientierten Kontext betrachtet und für die Einbeziehung jeglicher Funktionen ausgeweitet werden. Dementsprechend kann gemäß folgender Formel die Anzahl neuer Funktionen für jede der zu transformierenden Funktionen berechnet werden.

$$N = \frac{I}{E} \quad , \text{ mit } \quad I \in \mathbb{N}, E \in \mathbb{N}, E \leq I \quad \Rightarrow \quad N_{\max} = I, N_{\min} = 1$$

N : Anzahl neuer Funktionen

I : Anzahl der zu verlagernden Instruktionen

E : Einteilungsgröße

Die Einteilungsgröße kann durch eine konstante Anzahl an Instruktionen ausgedrückt werden. Wird als Einteilungsgröße ein Basic Block gewählt, entspricht E der Anzahl an Terminatoren in den zu transformierenden Funktionen. N_{\max} kann lediglich durch Hinzufügen von Instruktionen ohne Relevanz für die Programmlogik oder Spreizen von vorhandenen Instruktionen überstiegen werden, was nicht Gegenstand dieser Untersuchung ist. Bei $E = 1$ würde für jede Instruktion eine neue Funktion erstellt werden, sodass bezüglich der Zunahme von Funktionen das vorhandene Potential genutzt werden würde.

Zunahme von logischen Prädikaten

Unter einem logischen Prädikat wird vereinfacht das Ergebnis einer logischen Operation (*und, größer als, et cetera*) verstanden. Der Algorithmus sieht keine Erweiterung des Programms mit logischen Operationen vor. Dennoch kann bei näherer Betrachtung der Systemfunktion zum *Sperren eines Semaphors* die logische Evaluation des Zustands des als Argument übergebenen Semaphors bemessen werden. Die Formel zur Berechnung der absoluten Anzahl an hinzugekommenen Funktionen zum *Sperren eines Semaphors* ist identisch mit der Formel zur Berechnung der absoluten Anzahl neuer Funktionen.

Zunahme der Verschachtelung

Unter Verschachtelung wird ein Konstrukt verstanden, bei dem ein (durch ein logisches Prädikat) bedingter Ausführungsbereich einen weiteren bedingten Ausführungsbereich mit beliebig rekursiver Fortsetzung enthält. Wie auch bei den logischen Prädikaten hat die Transformation diesbezüglich keine Auswirkung, es kann sich jedoch aufgrund des geänderten Kontrollflusses über Semaphore eine empfundene Zunahme der Verschachtelung einstellen.

Zunahme der Anzahl der Methodenargumente

Die Anzahl der Methoden- beziehungsweise Funktionsargumente wird nicht modifiziert und die neu erstellten Funktionen erwarten keine Übergabewerte, sodass gemäß dieser Metrik die Wirksamkeit nicht erhöht wird.

Zunahme der Abhängigkeit zwischen Variablen von Klasseninstanzen und Zunahme der Höhe des Vererbungsbaums

Hinsichtlich objektorientierter Komponenten erfolgt keine Veränderung, sodass gemäß dieser Metrik die Wirksamkeit nicht erhöht wird.

Zunahme der Variablen, die in mehreren Basic Blocks genutzt werden

Im Grunde wird, sofern die gewählte Einteilungsgröße der zu verlagernden Funktionalität einem Basic Block entspricht, trotz der Umwandlung von lokalen Identifiern zu globalen Variablen die Anzahl der Basic Blocks, in denen der besagte Identifier verwendet wird, nicht verändert. Wird bei dieser Metrik jedoch die Entfernung der Variablenabhängigkeit miteinbezogen (wie auch in der am Anfang des Abschnitts referenzierten Literatur), so kann aufgrund des Speicherorts im globalen Raum und der Nutzung in verschiedenen Funktionen ein erheblicher Anstieg der Komplexität gemäß dieser Metrik festgestellt werden. Hinzu kommen neu erstellte globale Variablen für die Semaphore-Identifier und die Thread-Identifier, welche mehrfach in der ursprünglichen Funktion und den neuen Funktionen referenziert werden.

Um den in diesem Kontext charakteristischen Wert zu erhöhen, kann die beschriebene Optimierung, bei der ein Identifier nur in den globalen Kontext verlagert wird, wenn dieser in einem anderen Basic Block genutzt wird, deaktiviert werden, wobei dies zu Lasten der Kosten gehen wird. Auch wenn die Einteilungsgröße der zu verlagernden Funktionalität kleiner als ein Basic Block gewählt wird, wird die Zunahme der Komplexität gemäß dieser Metrik erhöht.

Auf die Erstellung und Beschreibung einer Formel wird an dieser Stelle verzichtet, da es sich gemäß der dargelegten Definition zum einen um eine qualitative Zunahme handelt und zum anderen

die absolute quantitative Zunahme (simplifiziert) der zweifachen²⁰ Anzahl neuer Funktionen entspricht.

Die folgenden Metriken wurden [84] entnommen:

Längste gemeinsame Teilsequenz²¹

Bei der längsten gemeinsame Teilsequenz (englisch: longest common subsequence; Kürzel: LCS) ist es im Gegensatz zur längsten gemeinsamen Teilzeichenfolge unerheblich, ob die gemeinsamen Zeichen direkt aufeinanderfolgen. Die längste gemeinsame Teilsequenz bildet sich aus der maximalen Anzahl an Zeichen, die in den betrachteten Sequenzen bezüglich ihrer relativen Folge gleich sind. Beispiel: Die längste gemeinsame Teilsequenz der Zeichenfolgen „abcabc“ und „aabbcc“ ist „abbc“. Im Vergleich ist die längste gemeinsame Teilzeichenfolge „ab“ oder „bc“. [85]

Anhand dieses Werts lassen sich Aussagen zur Ähnlichkeit zwischen zwei Dateien und folglich zum Ausmaß der Umwandlung treffen. Für die Berechnung der Ähnlichkeit wird die folgende Formel verwendet:

$$\text{Ähnlichkeit} = \frac{\text{Länge}(LCS(Seq_1, Seq_2))}{\sqrt{\text{Länge}(Seq_1) * \text{Länge}(Seq_2)}} \quad [86]$$

Die Auswertung der Messdaten kommt zu folgendem Ergebnis: Die prozentuale Ähnlichkeit bewegt sich zwischen 23,02 % und 50,24 % bei einem Mittelwert von 39,11 %²². Es fällt auf, dass Programme mit einer hohen Zunahme der Programmgröße geringere Ähnlichkeitswerte aufweisen und *vice versa*. Dies lässt sich durch das Gewicht der Größe des Programms im Divisor der verwendeten Formel begründen. Obwohl die Ähnlichkeit zurecht durch extremere Größenverhältnisse abnimmt, soll zusätzlich die Ähnlichkeit unter Bezug der größten gemeinsamen Teilsequenz auf lediglich die Länge des nicht obfuskierten Programms betrachtet werden (ähnlich wie in [84]), um den beschriebenen Einfluss zu verringern. Obwohl die Ähnlichkeit hierdurch auf 75,52 % ± 10 % ansteigt, ist kein eindeutiger Zusammenhang mit der Zunahme der Programmgröße mehr erkennbar. Zusätzlich liegen die Ähnlichkeitswerte mit einer Spannweite von circa 10 % im Gegensatz zu den fast 30 % mit ersterer Formel dichter beieinander, sodass dieser Wert für Vergleiche im Bezug auf Obfuskationstechniken gegebenenfalls zu bevorzugen ist. Wird die Ähnlichkeit nur auf die Code-sektion bezogen, verringern sich die ermittelten Mittelwerte jeweils um etwa 10 %.

Zum Vergleich dieser Werte wurde die Ähnlichkeit zwischen den vergleichbaren Programmen xz und zip berechnet²³. Bei der ersten Formel liegt die Ähnlichkeit mit 35,34 % leicht unter dem Mittelwert der Testprogramme aus dem Coreutils Softwarepaket. Bei der zweiten Formel wird die längste gemeinsame Teilsequenz auf die kleinere Programmgröße bezogen. Der Wert beträgt 59,53 % und liegt somit deutlicher unter dem ermittelten Mittelwert. Hierbei ist jedoch zu beachten, dass auch das Größenverhältnis der beiden Programme mit einer fiktiven Zunahme von 183,74 % deutlich geringer ist. Zusätzlich wurden die Ähnlichkeit zwischen den zwei verschiedenen Programmen du und scp berechnet²⁴, welche jedoch eine nahezu identische Programmgröße aufweisen. Durch die

20 Thread- und Semaphore-Variable

21 siehe Anhang 7.2 Messdaten zur längsten gemeinsamen Teilsequenz und 7.3 Messdaten zur längsten gemeinsamen Teilzeichenfolge

22 Bei der 32-bit Variante fallen diese Werte noch etwas geringer aus.

23 xz (Version 5.2.2) und zip (Version 3.0) einer Ubuntu bionic Distribution

24 du (Version 8.28) und zip (Version 7.6p1) einer Ubuntu bionic Distribution

ähnliche Größe produzieren beide Formel einen Wert von ungefähr 41,7 %. Auch ein Vergleich eines mit GCC kompilierten bzip2-Programms mit einem durch Clang kompiliertem führt bei beiden Formeln zu etwa 42 %.

Dementsprechend kann konstatiert werden, dass die Ähnlichkeit mit der längsten gemeinsamen Teilsequenz stark durch den Größenunterschied der Programme beeinflusst wird, was der Thread-basierten Obfuskation immanent ist. Gleichwohl indiziert der Vergleich der Werte mit den Werten beliebiger anderer Programmpaare, dass aufgrund der Überschneidung des Wertebereichs keine eindeutige Zugehörigkeit eines obfuskierten Programms zu einem nicht obfuskierten Programm anhand der längsten gemeinsamen Teilsequenz möglich ist.

Zuletzt kann auch die bereits erwähnte, längste gemeinsame Teilzeichenfolge als Indikator für einen Zusammenhang zwischen zwei Dateien dienen. Diese fällt bei der Anwendung auf ein nicht obfuskiertes Sample und ein mit der Thread-basierten Obfuskation transformiertes Sample häufig deutlich höher aus als bei zwei beliebigen Samples. Dies ist vornehmlich von anderen Sektionen abhängig, insbesondere von der Sektion, welche die konstante Daten wie Zeichenfolgen zur Ausgabe für den Nutzer enthält. Bei Anwendung auf die Codesektion ist der Wert dieser Metrik bei der Obfuskation jedoch vergleichbar mit beliebigen Samples, sofern keine Funktionen von der Transformation ausgenommen wurden. Im Hinblick auf die Coreutils liegt die längste gemeinsame Teilzeichenfolge für alle Paarungen bei 896 B und kommt durch die rekursive Funktion *quotearg_buffer_restyled* zustande.

Abschließend gilt es zu berücksichtigen, dass im Hinblick auf Maschinencode die Semantik bei loser Betrachtung einer Bytefolge nicht gut genug repräsentiert wird, weshalb die Anwendung dieser Metrik auf die disassemblierte Darstellung geeigneter wäre. Doch auch hier entstehen beim semantischen Vergleich Schwierigkeiten durch die Nutzung von relativen und absoluten Adressen, die sich bei einer Transformation ändern können, sodass diesbezüglich zusätzliche Maßnahmen notwendig werden. In [87] werden 61 Ansätze für die Bestimmung der Ähnlichkeit von binärem Code untersucht, weshalb aufgrund dieser Dimension das Thema hier nicht fortgeführt werden soll.

Zunahme der Instruktionen²⁵

Die Zunahme an Instruktionen ist mit der Zunahme der Programmgröße vergleichbar. Auf die Erstellung und Beschreibung einer Formel soll an dieser Stelle verzichtet werden, da die Praktikabilität der Formel durch ihre Komplexität aufgrund etlicher Abhängigkeiten (zum Beispiel Unterschied Windows/Linux und Bedingungen wie „Rückgabewert vorhanden“) nicht gegeben wäre. Hinzu kommt die nicht beherrschbare Varianz bei der Generierung des Maschinencodes (siehe auch Unterabschnitt „Datenfluss“).

Wie erwartet, ist das Diagramm der Zunahme der Instruktionen über die Anzahl der Instruktionen im nicht obfuskierten Programm optisch fast identisch mit dem Diagramm in Abbildung 3. Marginale Differenzen können durch die Abhängigkeit vom Größenverhältnis der Codesektion zu anderen Sektionen entstehen. Zudem fällt auf, dass die Werte auf der X-Achse kontinuierlicher sind als die Programmgröße, weil bei der Anzahl der Instruktionen keine Ausrichtung wie bei den Sektionen erfolgt. Konkret steigt die Anzahl der Instruktionen im Mittel um das fünf- bis sechsfache, was nach-

²⁵ siehe Anhang 7.4 Messdaten zur Zunahme der Instruktionen

vollziehbar etwas mehr als die Zunahme der Gesamtgröße ist, weil die Codesektion am stärksten von der Obfuskation betroffen ist.

Da Diagramme wie bereits beschrieben identisch sind, soll an dieser Stelle die graphische Darstellung der Anzahl der Instruktionen im obfuskierten Programm in Abhängigkeit von der Anzahl der Instruktionen im nicht obfuskierten Programm betrachtet werden (Abbildung 4). Wie zu erwarten war, verläuft der Graph linear mit einem Anstieg entsprechend der bereits geschilderten prozentualen Zunahme. Bemerkenswert ist hierbei jedoch, dass zwei anscheinend unabhängige, lineare Graphen mit unterschiedlichem Versatz zu sehen sind, wodurch sich die Programme in zwei unterschiedliche Kategorien teilen ließen. Der konkrete Grund für diese besondere Auffälligkeit konnte vorerst nicht eruiert werden, weil zumindest beim Quellcodevergleich keine außergewöhnlichen Eigenschaften festgestellt werden konnten²⁶. Dennoch wird, wie bei der Zunahme der Programmgröße beschrieben, vermutet, dass der Grund für diesen Versatz mit steigender Programmgröße beziehungsweise Anzahl der Instruktionen kompensiert wird und sich demnach die Graphen zusammenschließen.

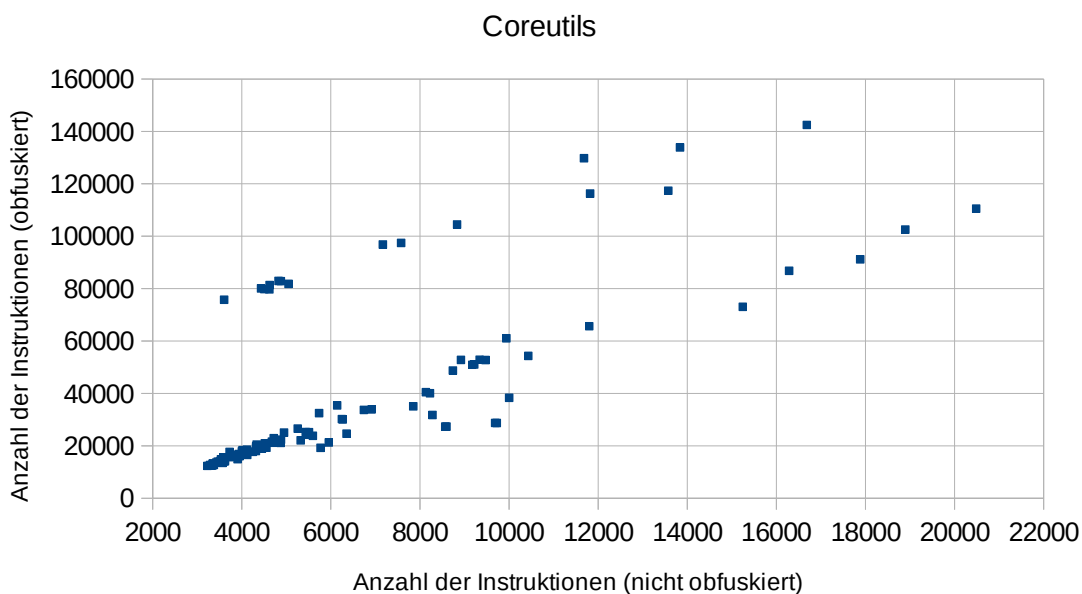


Abbildung 4: Zunahme der Instruktionen

Zunahme von Verzweigungsinstruktionen

Die Zunahme von Verzweigungsinstruktionen führt zur Steigerung der Komplexität und steht im Zusammenhang mit logischen Prädikaten und der Verschachtelung. Wie bereits aufgezeigt, werden durch die Transformation keine neuen Verzweigungsinstruktionen erzeugt, weshalb gemäß dieser Metrik die Wirksamkeit nicht erhöht wird.

4.2.2 Widerstandsfähigkeit

Die Widerstandsfähigkeit einer Obfuskationsmethode soll angeben, wie komplex der Umkehrprozess dieser Methode ist und wie hoch somit der Aufwand der Deobfuskation ausfällt. Unterschieden wird hierbei zwischen dem Aufwand, der durch einen Programmierer bei der Erstellung des Deob-

²⁶ Auch der Einfluss durch eine untypische Verteilung rekursiver Funktionen konnte ausgeschlossen werden.

fuskators erforderlich ist, und den Ressourcen, die bei der Rücktransformation durch einen Computer aufgewendet werden müssen. Das folgende Koordinatensystem wurde hierbei vorgestellt:

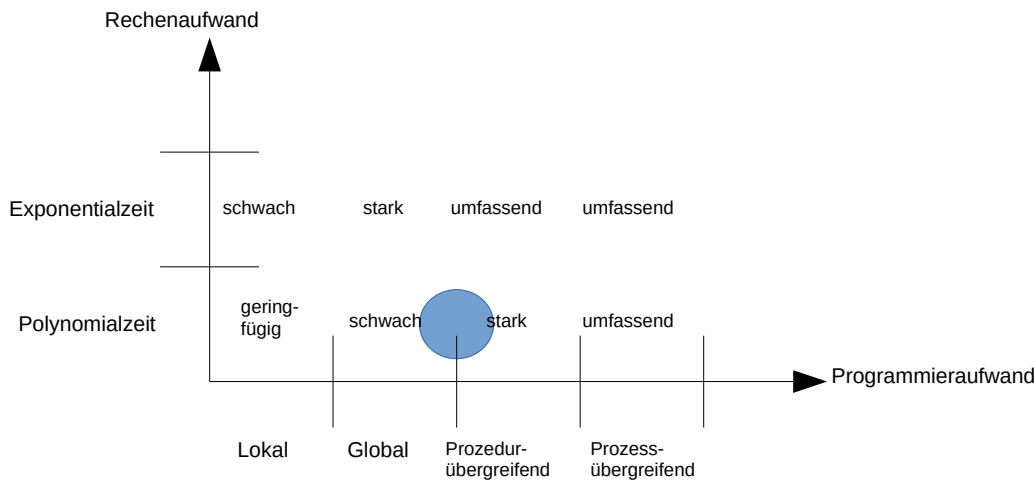


Abbildung 5: Koordinatensystem zur Einordnung der Widerstandsfähigkeit der Thread-basierten Obfuskation [2]

Das Maß der Widerstandsfähigkeit reicht von *geringfügig* über *stark* bis *irreversibel*. Irreversibilität kann nur durch das Entfernen von Informationen erreicht werden. Bei der vorliegenden Implementierung geht lediglich die Information verloren, ob es sich bei einer Variable um eine lokale oder globale handelte. Für die Deobfuskation und die anschließende Analyse ist diese Information jedoch als irrelevant einzustufen, sodass die Reversibilität gegeben ist.

Um den Aufwand des Deobfuskationsprozesses einschätzen zu können, muss zunächst das Ziel der Deobfuskation definiert und der Rückwandlungsvorgang schematisch dargestellt werden. Das Ziel der Deobfuskation ist im optimalen Fall die exakte Wiederherstellung des ursprünglichen Codes. Dies ist mit Ausnahme der Irreversibilität des Speicherorts der Variablen annähernd möglich. Der Deobfuskationsprozess könnte gemäß folgendem Ansatz erfolgen:

- Identifikation von transformierten Funktionen
- Feststellung der Ausführungsreihenfolge der durch diese Funktion aufgerufenen Funktionen anhand der Semaphore
- Verschieben der Instruktionen aus den neuen Funktionen in die ursprüngliche Funktion und Anpassung der Terminatoren gemäß festgestellter Ausführungsreihenfolge
- Entfernen aller überflüssigen Funktionen und Instruktionen.

Weitere Gegenmaßnahmen beziehungsweise eine gesteigerte Detailtiefe werden/wird im Abschnitt „Ansätze für Gegenmaßnahmen“ präsentiert.

Wie am Koordinatensystem in Abbildung 5 zu erkennen ist, wird der Rechenaufwand in Polynomialzeit $O(n^k)$ und Exponentialzeit $O(e^n)$ differenziert, wobei sich bezüglich der Thread-basierten Obfuskation n an der Anzahl der transformierten Funktionen in Kombination mit dem jeweiligen Umfang und der Einteilungsgröße orientiert. Da bei der Durchführung der Deobfuskation gemäß skizziertem Ansatz keine Probleme der Klasse NP oder höher erkennbar sind, ist davon auszugehen, dass die Deobfuskation in nicht höher als polynomieller Zeit lösbar ist. Neben des Zeitaufwands

kann auch das nötige Speicherplatzvolumen in die Betrachtung einfließen. Aber auch hier lässt der doch sehr geradlinige Ansatz den Schluss zu, dass von nicht mehr als einem linearen Anstieg des benötigten Speicherplatzes in Abhängigkeit von der Intensität der Obfuskation auszugehen ist.

Bezüglich des Programmieraufwands erfolgt die Skalierung anhand des Ausmaßes des Transformationsprozesses gemäß folgender Kriterien:

- **Lokal:** Betrifft nur einen Basic Block.
- **Global:** Betrifft einen gesamten Kontrollflussgraph.
- **Prozedurübergreifend:** Betrifft den Informationsfluss zwischen Prozeduren.
- **Prozessübergreifend:** Betrifft die Interaktion zwischen einzelnen Programmfäden.

Die Obfuskationsmethode verschiebt Funktionalität in neue Funktionen die als Threads gestartet werden, sodass sich dies auf den gesamten Kontrollflussgraph einer Funktion auswirkt. Entsprechend kann die Methode mindestens als Obfuskation mit globalen Auswirkungen eingeordnet werden. Auch prozedurübergreifend modifizierende Eigenschaften können der Transformation zugeschrieben werden. Wenn unter einer Prozedur eine Funktion verstanden wird, so werden zum einen die Funktionsparameter entfernt, sodass die nicht mehr als solche erkennbaren Argumente über globalen Variablen übergeben werden. Zum anderen können auch einzelne Basic Blocks als Prozedur verstanden werden, deren Datenaustausch nach der Transformation auch über globalen Variablen erfolgt sowie deren Kontrollfluss über Semaphore anstelle von unmittelbaren Terminatoren (also Sprunginstruktionen und Return-Instruktionen) realisiert wird. Prozessübergreifend modifizierende Eigenschaften liegen hingegen nur vor, sofern bereits vorhandene Parallelität obfuskiert wird. Zwar werden in der Implementierung für diese Arbeit Threads erstellt, welche aufgrund der erheblichen Dominanz die Analyse erschweren können. Falls überhaupt Threads im Originalprogramm vorhanden sind, wird jedoch zum einen nicht unmittelbar Einfluss auf die Interaktion zwischen urchinlichen Threads genommen und zum anderen ist die Identifikation von urchinlichen Threads weiterhin möglich. Demgemäß stellt der blau markierte Bereich in Abbildung 5 die Einordnung des in dieser Arbeit erstellten Obfuskationsalgorithmus dar.

Über diese Betrachtungsweise hinaus ist ein hervorzuhebender Vorteil der vorgestellten Obfuskation, dass ursprüngliche Funktionen durch die Transformation in einem Ausmaß verändert werden, wodurch die Anwendung von generischen Deobfuskatoren unmöglich wird. Würden die neu erstellten Funktionen gewöhnlich (also nicht als Thread) aufgerufen werden, könnte dieser als *Outlining* beschreibbare Vorgang mit einer simplen Compileroptimierung *Inlining* rückgängig gemacht werden. Es muss stattdessen eine spezifische Deobfuskationsroutine angewandt werden, die zudem erst nach der Analyse des Codes auf Anwendbarkeit eingesetzt werden kann. Minimale Abweichungen in der vorgestellten Implementierung können dazu führen, dass die Routine angepasst werden muss, wodurch sich der Aufwand erhöht. Hinzu kommt, dass die Deobfuskation von nativ ausführbaren Dateien (beziehungsweise Maschinencode im Allgemeinen) im Zusammenhang mit den bereits erläuterten Schwierigkeiten beim Binary Rewriting die dynamische Analyse beschränkt.

Wie in Abbildung 5 dargestellt, kann die Widerstandsfähigkeit der Thread-basierten Obfuskation lediglich als *schwach* bis *stark* eingestuft werden. Hierzu sollte jedoch beachtet werden, dass individuelle Obfuskationstechniken aufgrund einer spezifischen Zielsetzung einen beschränkten Auswir-

kungsbereich haben und somit selten eigenständig eine *starke* bis *umfassende* Widerstandsfähigkeit aufweisen dürfte. Auch die Datenflussphase könnte als eigenständiger Obfuskationsprozess bezeichnet werden. Dieser wird bei der Betrachtung jedoch miteinbezogen, da es sich bei diesem Schritt um eine zwangsläufige Voraussetzung für die nachfolgende Transformation handelt. Aus diesem Grund ist zu betonen, dass sich durch Erweiterung beziehungsweise Kombinationen einer weiteren (gegebenenfalls grundlegenden) Obfuskationstechnik mit der Thread-basierten Obfuskation die Widerstandsfähigkeit mit geringem Aufwand in positive Richtung auf beiden Achsen verschieben und auf *umfassend* anheben lässt. Auf die diesbezüglich zur Verfügung stehenden Möglichkeiten und Implikationen soll in den Abschnitten „Kombination mit anderen Obfuskationsmethoden“ und „Weitere Optimierungen“ genauer eingegangen werden.

4.2.3 Kosten

Die Kosten einer Obfuskationstechnik können anhand der benötigten Zeit für die Transformation, den höheren Speicherplatzbedarf und durch die Steigerung der Ausführungsdauer qualifiziert und auch quantifiziert werden. Hierbei wird zur Qualifizierung ebenfalls eine Unterteilung des Maßstabs anhand von Komplexitätsklassen vorgenommen:

- konstanter Ressourcenzuwachs $O(1)$: kostenlos
- linearer Ressourcenzuwachs $O(n)$: billig
- polynomieller Ressourcenzuwachs $O(n^k)$: kostspielig
- exponentieller Ressourcenzuwachs $O(e^n)$: teuer

Da diese Komplexitätsklassen das Verhältnis der benötigten Ressourcen von obfuskiertem Programm und nicht obfuskiertem Programm darstellen sollen, wurden entsprechend praktische Messungen vorgenommen.

Die Zeitmessungen wurden auf einem Intel Pentium Silver N5000 mit Hilfe des Linux Kommandos *perf* [88] durchgeführt. Hierbei wurden folgende Empfehlungen für den jeweiligen Prozess angewandt: Die Scheduling Priorität wurde auf den höchsten Wert gesetzt, es wird die FIFO-Strategie angewandt, es wird geprüft, ob kein Kontextwechsel während der Ausführung stattgefunden hat, die Frequenz der CPU wurde auf einen konstanten Wert von 1,1 GHz gesetzt und der Prozess wird an einen einzigen Prozessor gebunden [89]. Es ist zu beachten, dass sich die Ausführungsdauer des jeweiligen Samples durch die Nutzung von *perf* und der Bindung an einen einzigen Prozessor erhöht.

Transformationsdauer²⁷

Die Transformation ist mit einem Optimierungsdurchlauf eines Compilers zu vergleichen. Da keine intensiven Berechnungen notwendig sind, sollte das Verhältnis zwischen reinem Kompiliervorgang und Kompiliervorgang mit Transformation nicht erheblich ausfallen. Für die Messungen wurden die Quelldateien des Coreutils Softwarepakets ohne den Linkvorgang kompiliert. Für jede Quelldatei wurden zehn Durchgänge durchgeführt und der arithmetische Mittelwert gebildet. Zudem wurden die Werte der Messreihe mit denen einer zweiten erstellten Messreihe verglichen, um die Konsistenz der Werte zu verifizieren.

²⁷ siehe Anhang 7.5 Messdaten zur Transformationsdauer

Als Ergebnis liegt eine recht gleichmäßige Verteilung der Zunahme der Transformationsdauer vor. Zwischen einer minimalen Zunahme von 12,62 % und einer maximalen Zunahme von 112,07 % liegt der Mittelwert bei 52,70 %. Ein konkreter Zusammenhang zwischen der Größe des kompilierten Moduls (grob gemessen in Anzahl der Funktionen) und der relativen Zunahme konnte nicht festgestellt werden; leichte Tendenzen sind erkennbar (siehe Abbildung 6). Es ist davon auszugehen, dass ein Großteil der zusätzlich benötigten Zeit durch die Erkennung von rekursiven Funktionen und Basic Blocks mit multiplen Durchläufen entsteht und demnach auf die Komplexität, insbesondere die Verschachtelung der einzelnen Funktionen zurückzuführen ist. Als Konklusion wird dementsprechend vielmehr ein linearer als polynomieller Ressourcenzuwachs postuliert.

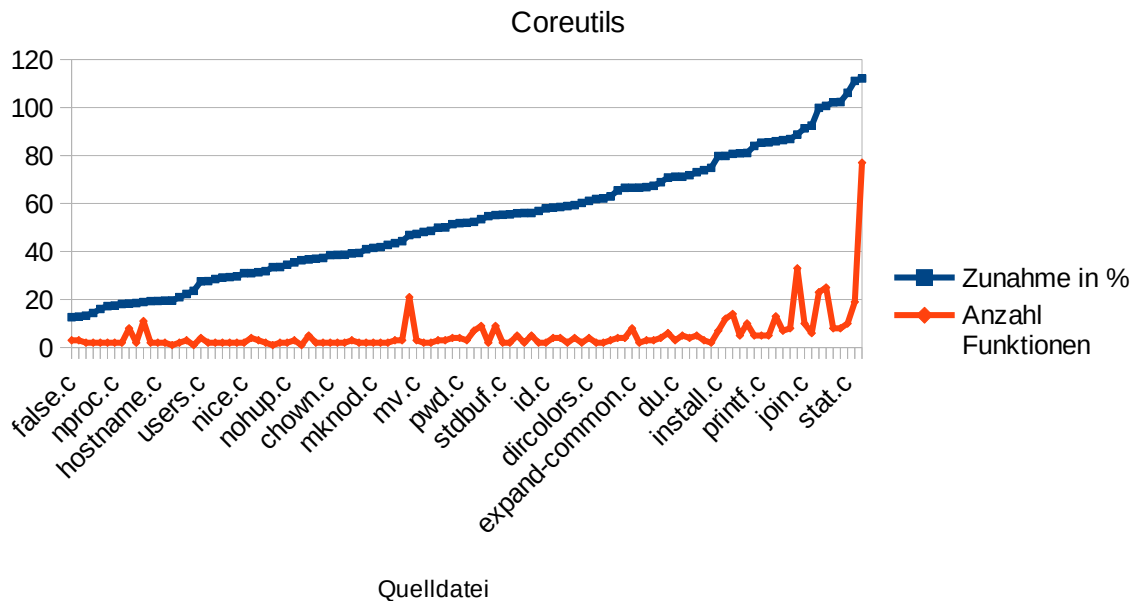


Abbildung 6: Zunahme der Übersetzungszeit bei Anwendung der Obfuskation

Schließlich erfolgte noch die Ermittlung der Zeit, die für den gesamten *make*-Prozess des Softwarepakets benötigt wird. Das Ergebnis mit circa 300 s für die Durchführung ohne Obfuskation und 458 s mit Obfuskation bestätigt das bereits dargestellte Ergebnis. Letztlich ist auch zu beachten, dass durch die Generierung größerer Objektdateien auch die benötigte Zeit beim Linkvorgang steigt. Diese Zunahme sollte jedoch marginal ausfallen.

Speicherplatz²⁸

Es steigt sowohl der statische (also die Größe des Codes auf der Festplatte) als auch der dynamische (also die Kapazität des Prozessspeichers während der Ausführung) Speicherplatzbedarf. Die Zunahme der Programmgröße wurde bereits in bei den Metriken für die Wirksamkeit berechnet und steht im direkten Konflikt mit dieser Metrik. Der Prozessspeicherbedarf wächst durch den zusätzlichen Speicher, der durch die Threads benötigt wird, sowie durch die notwendigen Verwaltungsinformationen. Ein relevanter Bedarfsanstieg des Prozessspeichers hinsichtlich der Ausführungsdauer des Prozesses konnte nicht festgestellt werden.

²⁸ siehe Anhang 7.6 Messdaten zum Speicherplatz

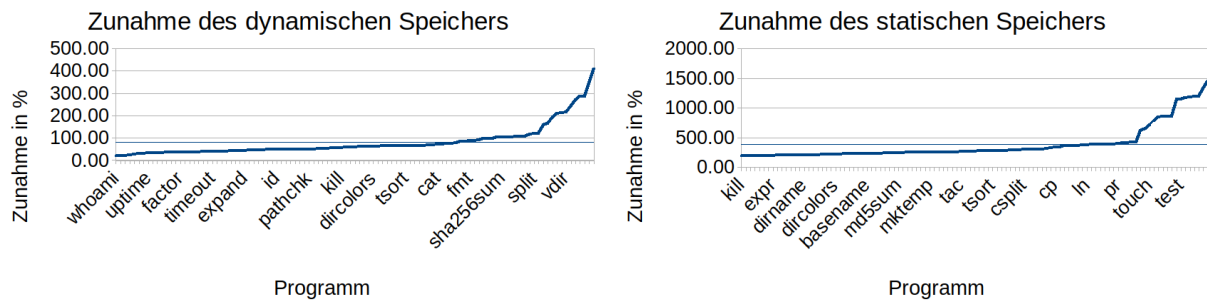


Abbildung 7: Gegenüberstellung der Zunahme des dynamischen und statischen Speichers

Um den Prozessspeicher zu bestimmen, wird die vom Linux *time* Kommando ausgegebene „maximum resident set size of the process during its lifetime“ (deutsch: maximale Größe des belegten Speichers des Prozesses im Arbeitsspeicher während des Ausführung) verwendet [90]. Die Messung wurde für jedes Programm zehnmal mit nachfolgender Bildung des arithmetischen Mittelwerts durchgeführt. Ebenso erfolgte die Verifizierung der Konsistenz der Werte durch Bildung einer zweiten Messreihe. Die Programme werden ohne Argumente gestartet und nach einer Sekunde terminiert, falls sie noch aktiv sind. Hierbei ist zu beachten, dass das Verhalten eines Programms eine ausgebildete Abhängigkeit von den Eingabeparametern aufweisen kann, sodass die ermittelten Werte keinesfalls das Maximum darstellen, durch die entsprechende Programmanzahl jedoch eine gebräuchliche Repräsentativität aufweisen.

In Abbildung 7 sind die Werte der Zunahme des dynamischen Speicher denen des statischen Speicher gegenübergestellt. Auf der linken Seite sind Werte dargestellt, die mit dem im vorigen Absatz beschriebenen Vorgehen ermittelt wurden. Auf der rechten Seite sind die bereits mit Abbildung 3

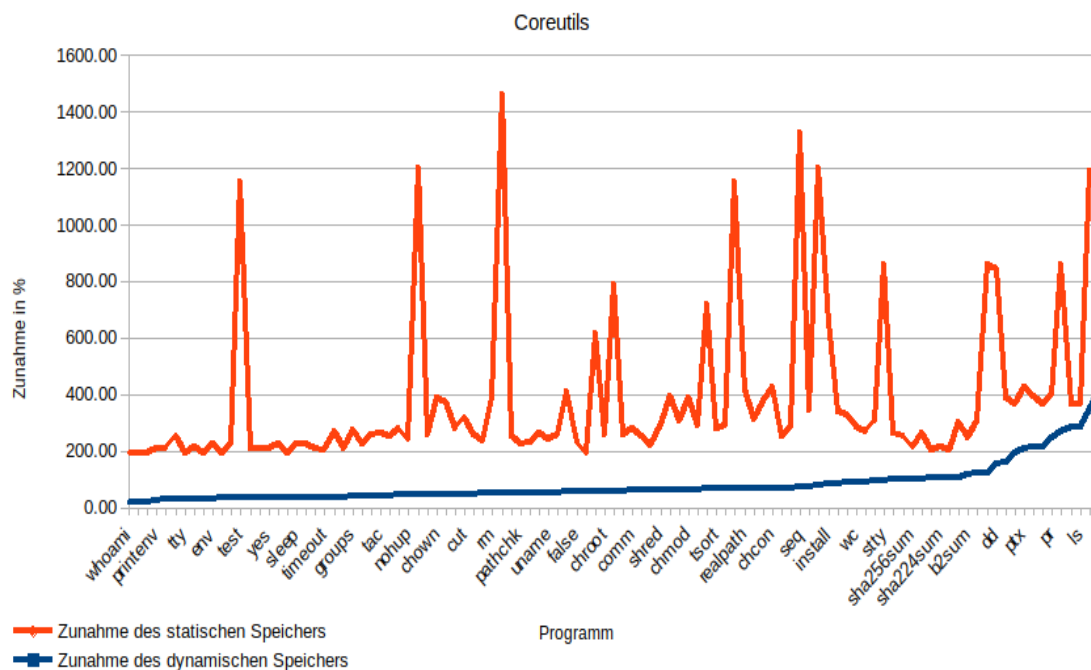


Abbildung 8: Zunahme des statischen Speichers unter Berücksichtigung der Ordnung der Zunahme des dynamischen Speichers

vorgestellten Werte zu sehen, hier jedoch ohne einen Bezugswert. Beim dynamischen Speicher liegt die Zunahme mit dem Maximalwert 410,88 % und dem Minimalwert 20,55 % im Mittel bei 81,27 %. Im Vergleich steigt der dynamische Speicher weniger stark an, ist jedoch mit einem fast verdoppelten Bedarf nicht vernachlässigbar. Wie bereits bei den Ergebnissen zur Wirksamkeit festgestellt, weichen auch beim dynamischen Speicher 10 bis 15 % stark vom eigentlichen Mittelwert ab.

Obwohl eine Ähnlichkeit bei Programmgröße und benötigtem Arbeitsspeicher naheliegend ist und die graphische Darstellung der Werte dies suggeriert, wird dieser annehmbare Zusammenhang bei der Vereinigung der beiden Koordinatensysteme aufgelöst (siehe Abbildung 8). Trotzdem treffen die im Unterabschnitt „Wirksamkeit“ getätigten Erwägungen unter Berücksichtigung dieser weiteren beeinflussenden Eigenschaften (neben der Abhängigkeit von den Argumenten) auch bezüglich der hier betrachteten Metrik zu. Folglich wird bei dieser Metrik von einem linearen Ressourcenzuwachs ausgegangen.

Performanz²⁹

Hinsichtlich der Performanz ist durch die zusätzliche Verwaltungsarbeit durch das Betriebssystem ein deutlicher Geschwindigkeitsverlust zu erwarten. Um diesbezüglich aussagekräftige Ergebnisse zu generieren, eignet sich die Bestimmung der Ausführungsdauer eines rechenintensiven Programms in obfuskiert und nicht obfuskiert Form, sodass diese Zahlen anschließend ins Verhältnis gesetzt werden können.

Als Referenzprogramme sollen die BBP Formel und bzip2 verwendet werden. Zwar werden bei beiden Programmen intensive Berechnungen durchgeführt, jedoch unterscheiden sich die Programme stark hinsichtlich ihrer Komplexität. Das Programm für die BBP Formel hat in 149 Codezeilen sechs Funktionen, von denen eine Funktion am Ende der Funktionsaufruffolge rekursiv ist. bzip2 hingegen hat 109 Funktionen verteilt auf 6419 Codezeilen. Durch diese starken Unterschiede bei der Komplexität sollen die möglichen Einflüsse der Obfuskation auf die Performanz stärker in der Breite untersucht werden.

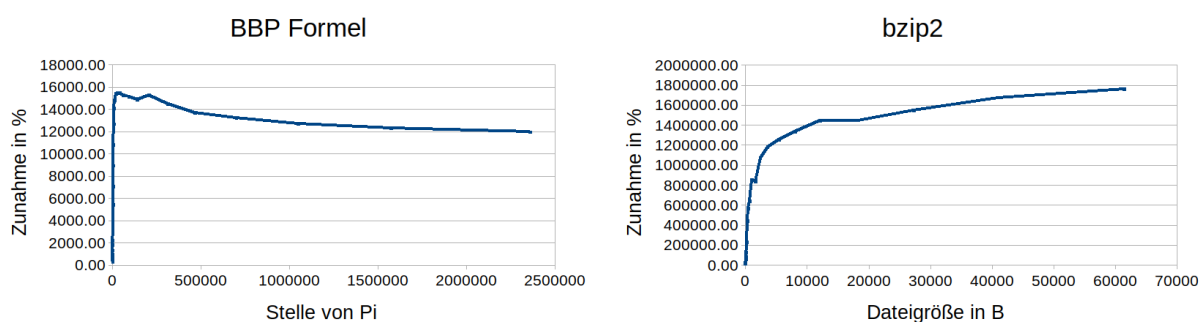


Abbildung 9: Gegenüberstellung BBP Formel und bzip2 hinsichtlich der Zunahme der Ausführungsdauer

Zunächst ist in in Abbildung 9 der Unterschied der beiden Referenzprogramme bei der Zunahme der Ausführungsdauer in Abhängigkeit von den Eingabeparametern zu sehen. Wie zu erkennen ist, streben die Graphen ab einem Argument, welches zu einer ausgedehnten Ausführungsdauer des

²⁹ siehe Anhang 7.7 Messdaten zur Performanz

Programms führt, einem konstanten Wert entgegen. Die Unterschiede beim Verhalten des Graphen vor diesem Wendepunkt kann zum einen auf Messungenauigkeiten zurückzuführen sein. Zum anderen können bei kleineren Ausführungszeiten bestimmte Eigenschaften des Programms sowie die Betriebshardware und -software zusätzlich das Ausführungsverhalten beeinflussen.

Als Unterschied der Programme ist hervorzuheben, dass bei der BBP Formel eine rechenintensive Routine, die in einer Schleife aufgerufen wird, aufgrund ihrer Rekursion nicht obfuskiert wurde. Bei bzip2 erfolgt die Kodierung der Eingangsdaten ebenfalls in einer Routine, welche in einer Schleife aufgerufen wird, welche aber durch die Obfuskation permanent neue Threads startet. Hierdurch lässt sich erklären, dass im Gegensatz zu bzip2 bei der BBP Formel ein leichter Abfall der Zunahme der Ausführungsdauer zu verzeichnen ist, weil der Zeitanteil und somit der Einfluss der nicht obfuskierten Funktion bei längerer Ausführungsdauer wächst. Auch hinsichtlich der Betriebsumgebung existieren Eigenschaften wie Cachegröße oder die Realisierung der Parallelisierung durch den Kernel, die einen Einfluss auf das Ausführungsverhalten haben und gesondert untersucht werden könnten.

Trotz der anscheinend konstanten Zunahme der Ausführungsdauer in Abhängigkeit von den Eingabedaten ist vielmehr die Zunahme in Abhängigkeit von der Programmgröße beziehungsweise Komplexität entscheidend. In Tabelle 2 wird neben dem Verhältnis der maximalen Zunahme auch das Verhältnis weiterer Indikatoren aufgezeigt, nämlich Codezeilen (englisch: Lines Of Code; Kürzel: LOC), Anzahl der (obfuskierten) Funktionen und die Programmgröße. Es ist zu erkennen, dass das Verhältnis dieser Indikatoren stark vom Verhältnis der Zunahme der Ausführungszeit abweicht. Trotz einiger Unwägbarkeiten, die bei dieser stichprobenartigen Betrachtung offen bleiben, muss somit von einem polynomiellen bis exponentiellen und daher kostspieligen bis teuren Zeitressourcenwachstums ausgegangen werden.

Programm	LOC	Anzahl Funktionen	Programmgröße in B (n. obfus.)	Max. Zunahme in %
BBP Formel	149	5	14664	15463,89
bzip2	6419	109	99728	1758444,95
Verhältnis	43,08	21,80	6,80	113,71

Tabelle 2: Verhältnis zwischen der BBP Formel und bzip2 hinsichtlich der Performanz

Bei Betrachtung der absoluten Zahlen scheint der Einsatz der Obfuskation praxisfern. Wohingegen für die Berechnung der 2.362.204. Stelle von Pi nur wenige Sekunden notwendig sind und die Komprimierung von 61.447 B in unter einer Sekunde erfolgt, benötigt die obfuskierte Variante mehr als zehn Minuten. Zum einen relativiert sich dies etwas, wenn die Restriktionen des Versuchsaufbaus entfallen, insbesondere die Bindung an nur einen Prozessor. Hierdurch verringert sich die Ausführungsdauer der obfuskierten Programme um circa 15 %, was auf die höhere Anzahl an zur Verfügung stehenden Prozessoren zurückzuführen ist. Zum anderen sollte sich vergegenwärtigt werden, dass als Zielgruppe eher Stapelverarbeitungsprogramme, Kommandozeileninterpreter oder Kommunikationssoftware vorgesehen ist. Es ist davon auszugehen, dass das wiederholte Aufrufen von Funktionen in Schleifen und das dadurch permanente Erstellen und Entfernen von Threads die Zeit im Kernelspace drastisch erhöht und am stärksten zu diesem Performanzverlust beiträgt. Der Umstand, dass diese Eigenschaft weniger stark bei der Zielgruppe ausgeprägt ist, und die Möglich-

keit, die Obfuskation nur auf selektierte Bereiche anzuwenden³⁰, erlauben trotz dieser hohen Kosten den praxistauglichen Einsatz.

Abschließend ist noch darauf hinzuweisen, dass die Threads in der umgekehrten Reihenfolge bezogen auf ihre eigentliche Anordnung in der nicht obfuskierten Form gestartet werden, deshalb die Abarbeitung der Funktion frühestens nach dem Start des letzten Threads beginnen kann und somit der ungünstigste Fall gemessen wurde. Außerdem wird angemerkt, dass bei der 32-bit Variante eine geringfügig bessere Performanz festgestellt werden konnte und dass bei Windows wegen den zusätzlichen Maßnahmen ein höherer Performanzverlust zu erwarten ist (siehe Unterabschnitt „Besonderheit beim Windows Betriebssystem“).

4.2.4 Zusammenfassung

„For Alice this means that she may have to choose between a highly efficient program, and one that is highly obfuscated. Typically, an obfuscator will assist her in this trade or by allowing her to choose between cheap and expensive transformations.“ [2]

An dieser Stelle soll eine qualitative Gesamtbewertung anhand der Ergebnisse der vorangegangenen Unterabschnitte formuliert werden.

Die Wirksamkeit wird in [2] als Verhältnis der Komplexität gemäß der betrachteten Metriken definiert. Da für ein Verhältnis mit unbegrenzten Operanden kein Maximum bestimmt werden kann, handelt es sich bei der Einordnung der Wirksamkeit in „niedrig“, „mittel“ oder „hoch“ vielmehr um eine qualitative Einschätzung. Hervorzuheben ist der Anstieg der Programmgröße mit einem vielfachen der ursprünglichen Programmgröße. Auch die geringe Ähnlichkeit zwischen obfuskiertem und nicht obfuskiertem Programm trägt zu einer hohen Wirksamkeit bei, jedoch entstehen Abzüge durch den Fokus auf die Codesektion. Darüber hinaus konnten bei den meisten Metriken zumindest bei entsprechender Betrachtungsweise qualitative Merkmale festgestellt werden, welche die Wirksamkeit erhöhen. Nur auf objektorientierte Eigenschaften entfaltet die Thread-basierte Obfuskation keine direkte Wirkung. Unter Betrachtung der genannten Gesichtspunkte lässt dies eine Einstufung der Wirksamkeit als „mittel“ mit positiven Tendenzen zu.

Die Widerstandsfähigkeit sowie die Kosten lassen aufgrund konkreterer Vorgaben eine einfachere Bewertung zu. Bezüglich der Widerstandsfähigkeit ist in Abbildung 5 bereits anhand der festgestellten Merkmale die Einordnung graphisch dargestellt. Zwar entstehen hinsichtlich der Transformationsdauer und dem zusätzlich benötigten Speicher keine gravierenden Nachteile, jedoch kann durch die starken Auswirkungen auf die Performanz die Obfuskation auch abseits von der quantitativen Betrachtung als kostspielig bis teuer eingestuft werden.

Bei der Betrachtung dieser drei metrischen Dimensionen in einem Spannungsdreieck, kann der Wirksamkeit am meisten Gewicht beigemessen werden (siehe Abbildung 10). Wohingegen die Widerstandsfähigkeit weniger stark in einem Spannungsverhältnis mit den anderen Metriken steht³¹, lässt sich die Potenz der Wirksamkeit beziehungsweise der Sparsamkeit als invertierte Kosten direkt durch die Parametrisierung der Funktionsselektion und Einteilungsgröße horizontal regulieren. Diese Faktizität entspricht der Kernaussage des Zitats am Anfang dieses Unterabschnitts, in

30 Wird nur die *main*-Funktion obfuskiert, ist bei beiden untersuchten Programmen ab einer (kleinen) Mindestausführungsdauer kein Unterschied messbar.

31 hierzu mehr im Abschnitt „Ansätze für Gegenmaßnahmen“

welchem zusätzlich die integrierte Unterstützung durch den Obfuskator in diesem Kontext nahegelegt wird.

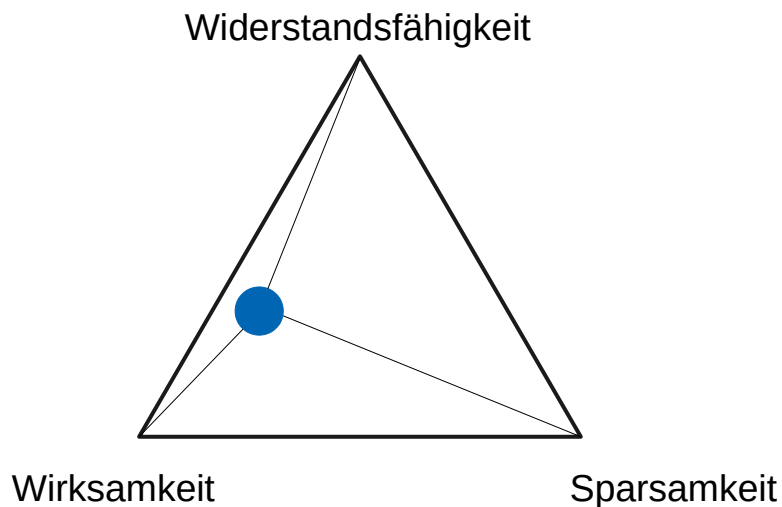


Abbildung 10: Spannungsdreieck der drei vorgestellten Obfuskationsmetriken

Letztendlich gilt es klarzustellen, dass der Geltungsbereich dieser Evaluation lediglich für Kommandozeilenprogramme mit übersichtlicher Komplexität gilt, dies jedoch auch die Zielgruppe darstellt. So kann der höhere Speicherplatzbedarf in diesem Kontext relativiert werden, weil bei den heutigen Dimensionen hinsichtlich verfügbarer Speicherplatzgrößen und Übertragungsraten für den Nutzer kein signifikanter Unterschied zwischen einer 100 KB oder einer 500 KB großen Datei besteht. Bei größeren Programmen wie Mimikatz wäre bei einem Anstieg von 1,6 MB auf 12 MB das Verhältnis zwischen Kosten und Nutzen abzuwägen. Dennoch ist zu beachten, dass diese Zunahme von 621,52 % nicht übermäßig über dem evaluierten Mittelwert von 386,33 % liegt und somit auch für größere Projekte zu gelten scheint.

Auch bei der Ausführungsdauer ist bei wenig rechenintensiven Aufgaben weiterhin kein Unterschied durch den Nutzer erkennbar. Außerdem ist bei interaktiven Programmen das Verhältnis zwischen Leerlauf des Programms während des Wartens auf die Nutzereingabe und der Ausführungsdauer – unabhängig ob obfuskiert oder nicht obfuskiert – verschwindend gering. So konnte bei der trivialen Bedienung von Mimikatz kein merklicher Unterschied zwischen nicht obfuskiert und obfuskiert Variante festgestellt werden.

Entsprechend der Erkenntnisse aus diesem Abschnitt folgt abschließend die Empfehlung, intensive Berechnungen eines Programms bei Anwendung der Obfuskation auszusparen.

4.3 Auswirkungen auf das Reverse Engineering

Die Auswirkungen auf das Reverse Engineering sind vielfältig und ebenso von einer gewissen Unbestimmtheit geprägt, da die Wirksamkeit von den individuellen Fähigkeiten eines (menschlichen) Betrachters abhängig ist [2]. Idealerweise müsste durch eine Nutzerstudie festgestellt werden, wie die obfuskierten Programme von erfahrenen Reverse Engineers aufgenommen werden, sodass mit diesen Daten aussagekräftigere Erkenntnisse bezüglich einer aggregierten, individuellen Wirksamkeit möglich sind. Für die Erstellung dieser Masterarbeit konnte aufgrund von Schwierigkeiten, die diese Methodik impliziert, die Durchführung nicht erfolgen. Stattdessen soll in diesem Kapitel die

gängige Herangehensweise bei der Analyse eines derart obfuskierten Samples betrachtet und etwaige durch die Obfuskation entstehenden Schwierigkeiten möglichst praxisnah beleuchtet werden.

Für die Prüfung, wie sich der Transformationsprozess auf das manuelle Reverse Engineering auswirkt, wurde die gebräuchliche, open-source und plattformunabhängige Reverse Engineering Software Ghidra verwendet, welche die am häufigsten benötigten Werkzeuge (wie Disassembler, Decompiler, Abhängigkeitsanalysator, Hexeditor) vereint. Kommerzielle Alternativen wie IDA Pro bieten mit einem integrierten Debugger einen Vorteil bei der dynamischen Analyse, für die Ziele dieser Arbeit ist jedoch die Nutzung eines separaten Debuggers (wie GDB [91] für Linux oder x64dbg [92] für Windows) ausreichend.

4.3.1 Statische Analyse

Um den Einfluss der Obfuskationstransformation zu präsentieren, sollen die Unterschiede zwischen nicht obfuskiertem Darstellung und obfuskiertem Darstellung eines Beispielprogramms anhand der Ausgaben von Ghidra visualisiert und erläutert werden. Als Beispielprogramm soll `bzip2` dienen, da die Komplexität der *main*-Funktion mit circa 250 Codezeilen hoch genug, aber trotzdem überschaubar ist und dementsprechend als geeignet befunden wurde. Während der Kompilierung wurde die Compileroptimierungsstufe 3 gewählt und alle (für die Ausführung des Programms unnötigen) Symbole entfernt, zum einen, um die Präsentation kompakter zu gestalten, und zum anderen, weil die Distribution von Software gewöhnlicherweise mit dieser Charakteristik erfolgt und die Analyse möglichst realitätsnah ausgerichtet werden soll. In Codeausschnitt 8 kann eine erste Impression des Unterschieds der *main*-Funktion in disassemblierter Darstellung zwischen obfuskiertem und nicht obfuskiertem Form des Beispiels gewonnen werden.

Am Anfang der Funktionen werden die in der Funktion benutzten, nicht volatilen Register mit der Instruktion *PUSH* auf dem Stack abgelegt [4]. Dementsprechend ist in der obfuskierten *main*-Funktion lediglich das Register *RBX* betroffen. Auf der linken Seite ist zu sehen, wie anschließend die Initialisierung von diversen globalen Variablen entsprechend dem Quellcode erfolgt. Auf der rechten Seite hingegen werden mit den ersten beiden folgenden Instruktionen die Funktionsargumente in dem dafür vorgesehenen globalen Speicherbereich abgelegt. Darauf folgt bereits die Initialisierung der einzelnen für die Synchronisierung benötigten Semaphore. Hierbei wird als erstes Argument der Zeiger auf den für dieses Semaphor vorgesehenen Speicherbereich übergeben (Register *EDI*), den beiden weiteren Argumenten, welche als *Attribut für die Erstellung von Parallelisierungsprimitiven* zu verstehen sind, wird der Wert *NULL* zugewiesen (siehe auch Unterabschnitt „Initialisierung“). Der Aufruf der Funktion *sem_init* in Verbindung mit den Registerzuweisungen wiederholt sich insgesamt 189-mal (dies entspricht auch der Anzahl der erzeugten Threads und der weiteren damit verbundenen Aufrufe). Sofern bei dem zu untersuchenden Programm keine hochgradige Parallelität erwartet wird, erscheint die Software aufgrund der ungewöhnlich hohen Anzahl an Systemaufrufen im Zusammenhang mit Parallelisierung direkt verdächtig.

Die Funktionalität des Anfangs der *main*-Funktion ist durchaus weiterhin im obfuskierten Programm enthalten. Aufzufinden ist sie in diesem konkreten Beispiel anhand des zweiten Arguments des letzten Systemaufrufs *pthread_create* in dieser Funktion, da für die Implementierung (wie beschrieben) die ursprüngliche Reihenfolge der Basic Blocks bei der Generierung der Systemaufrufe zur *Erstellung eines Thread* invertiert wurde. Die Schwierigkeit hierbei ist jedoch die Ausführungs-

reihenfolge der einzelnen als Thread gestarteten Funktionen zu bestimmen, besonders wenn die Reihenfolge der Systemaufrufe zufällig erzeugt wurde.

nicht obfuskiert

***** FUNCTION *****			

undefined	undefined FUN_00401490()	AL:1	<RETURN>
undefined4	Stack[-0x34]:4 local_34		

FUN_00401490			
00401490	55	PUSH	RBP
00401491	41 57	PUSH	R15
00401493	41 56	PUSH	R14
00401495	41 55	PUSH	R13
00401497	41 54	PUSH	R12
00401499	53	PUSH	RBX
0040149a	50	PUSH	RAX
0040149b	49 89 f6	MOV	R14,RSI
0040149e	89 fd	MOV	EBP,EDI
004014a0	48 c7 05	MOV	qword ptr [DAT_0061c6c8],0x0
	1d b2 21		
	00 00 00 ...		
004014ab	c6 05 12	MOV	byte ptr [DAT_0061c6c4],0x0
	b2 21 00 00		
004014b2	c6 05 23	MOV	byte ptr [DAT_0061c6dc],0x0
	b2 21 00 00		
004014b9	c6 05 4b	MOV	byte ptr [DAT_0061cf0b],0x0
	ba 21 00 00		
004014c0	c6 05 21	MOV	byte ptr [DAT_0061c6e8],0x1
	b2 21 00 01		
004014c7	c7 05 db	MOV	dword ptr [DAT_0061c2ac],0x0
	ad 21 00		
	00 00 00 00		
004014d1	c7 05 05	MOV	dword ptr [DAT_0061c6e0],0x9
	b2 21 00		
	09 00 00 00		
004014db	c6 05 18	MOV	byte ptr [DAT_0061c6fa],0x0
	b6 21 00 00		
004014e2	c6 05 21	MOV	byte ptr [DAT_0061cf0a],0x0
	ba 21 00 00		
004014e9	c7 05 dd	MOV	dword ptr [DAT_0061c6d0],0x0
	b1 21 00		

obfuskiert

***** FUNCTION *****			

undefined	undefined FUN_00401750()	AL:1	<RETURN>

FUN_00401750			
00401750	53	PUSH	RBX
00401751	89 3d 89	MOV	dword ptr [DAT_00624ee0],EDI
	37 22 00		
00401757	48 89 35	MOV	qword ptr [DAT_00624ee8],RSI
	8a 37 22 00		
0040175e	bf f0 4e	MOV	EDI=>DAT_00624ef0,DAT_00624ef0
	62 00		
00401763	31 f6	XOR	ESI,ESI
00401765	31 d2	XOR	EDX,EDX
00401767	e8 c4 fc	CALL	sem_init
	ff ff		
0040176c	bf 00 53	MOV	EDI=>DAT_00625300,DAT_00625300
	62 00		
00401771	31 f6	XOR	ESI,ESI
00401773	31 d2	XOR	EDX,EDX
00401775	e8 b6 fc	CALL	sem_init
	ff ff		
0040177a	bf 20 53	MOV	EDI=>DAT_00625320,DAT_00625320
	62 00		
0040177f	31 f6	XOR	ESI,ESI
00401781	31 d2	XOR	EDX,EDX
00401783	e8 a8 fc	CALL	sem_init
	ff ff		
00401788	bf 40 53	MOV	EDI=>DAT_00625340,DAT_00625340
	62 00		
0040178d	31 f6	XOR	ESI,ESI
0040178f	31 d2	XOR	EDX,EDX
00401791	e8 9a fc	CALL	sem_init
	ff ff		
00401796	bf 60 53	MOV	EDI=>DAT_00625360,DAT_00625360
	62 00		
0040179b	31 f6	XOR	ESI,ESI
0040179d	31 d2	XOR	EDX,EDX
0040179f	e8 8c fc	CALL	sem_init
	ff ff		

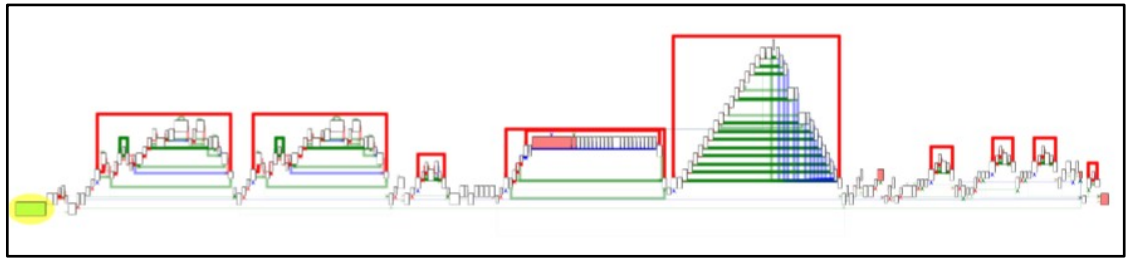
Codeausschnitt 8: Erster Eindruck zwischen obfuskiert und nicht obfuskiert main-Funktion

Wie bereits erläutert, ist die Transformation des Kontrollflusses (im Speziellen der graphischen Darstellung), sodass dieser für Analysezwecke nur noch bedingt zu gebrauchen ist, ein grundlegender Aspekt der vorgestellten Obfuskation. Für die diesbezügliche Wirksamkeit sind in Abbildung 11 die für die *main*-Funktion erzeugten Kontrollflussgraphen gegenübergestellt.³² Wie unschwer zu erkennen ist, wird das Ziel erreicht. Wohingegen aus dem Kontrollflussgraphen der nicht obfuskierten *main*-Funktion für das Reverse Engineering weiterführende Informationen abgeleitet werden können, handelt es sich bei der *main*-Funktion um einen einzigen Basic Block, der entsprechend strukturiert dargestellt wird. Der Kontrollflussgraph der neuen Funktionen besteht auch lediglich aus einem Basic Block, der die verlagerte Funktionalität beinhaltet, mit einer abschließenden Verzweigungsanweisung, deren Anzahl der Sprungziele effektiv identisch³³ mit der Anzahl der Nachfolger des ursprünglichen Basic Blocks ist. Ohne weiterführende Analyse der Semantik dieser Basic Blocks ist auch diese primitive Struktur wenig aussagefähig hinsichtlich des Aufbaus des Programms.

32 Die Graphen werden für eine kompaktere Darstellung waagrecht dargestellt.

33 Mit Ausnahme von Basic Blocks, die nur einen Nachfolger haben, in Verbindung mit aktivierten Compileroptimierungen. In diesem Fall wird die unbedingte Verzweigungsanweisung entfernt und es wird graphisch nur ein Basic Block dargestellt.

nicht
obfuskiert



obfuskiert

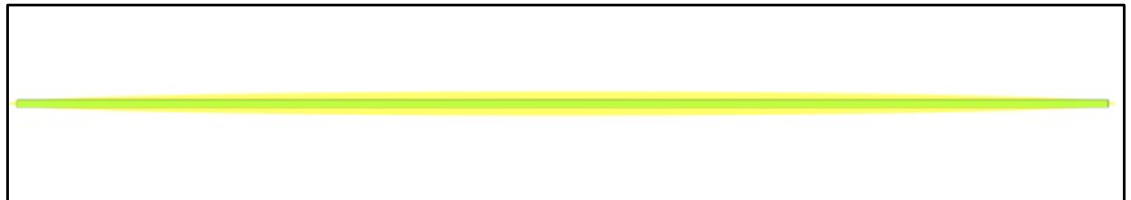


Abbildung 11: Gegenüberstellung des obfuskierten und nicht obfuskierten Kontrollflussgraphen

Eine weitere Möglichkeit um eine Vorstellung über die Struktur eines Programms zu gewinnen, ist die Visualisierung der durch eine Funktion aufgerufenen Funktionen. Diese als Funktionsaufrufgraph bezeichnete Darstellung ist für die *main*-Funktion des nicht obfuskierten Programms in Abbildung 12 zu sehen.

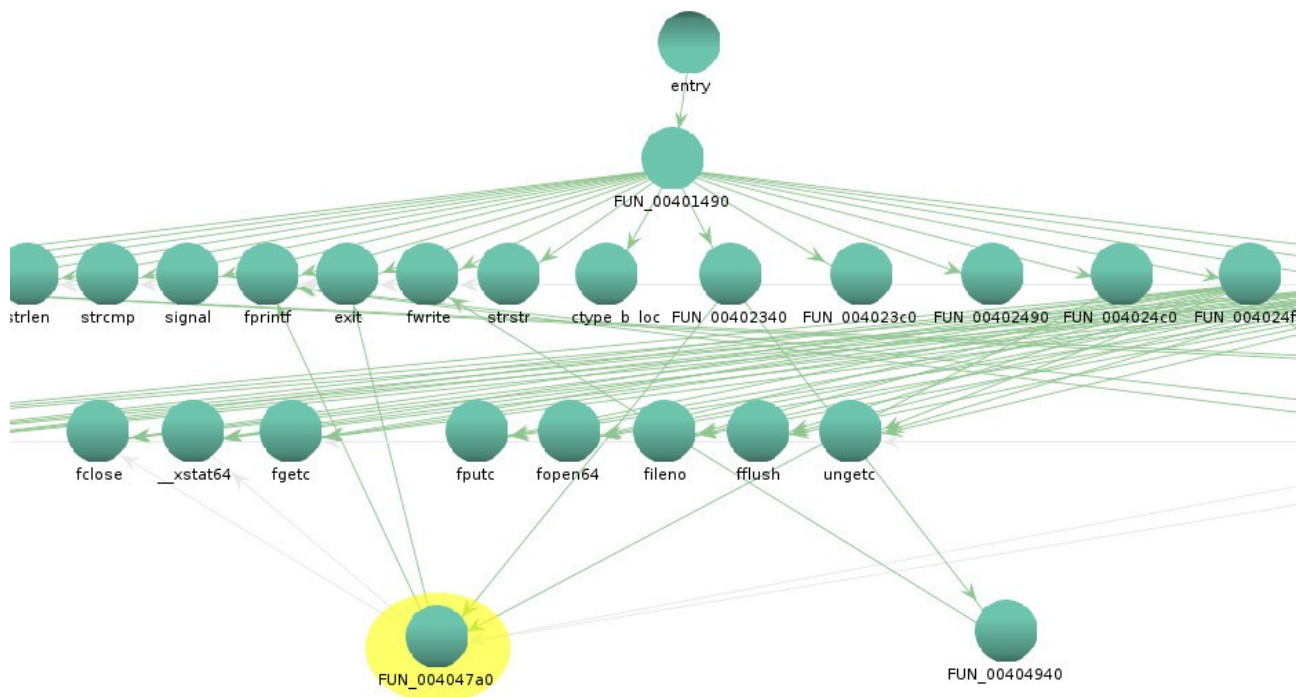


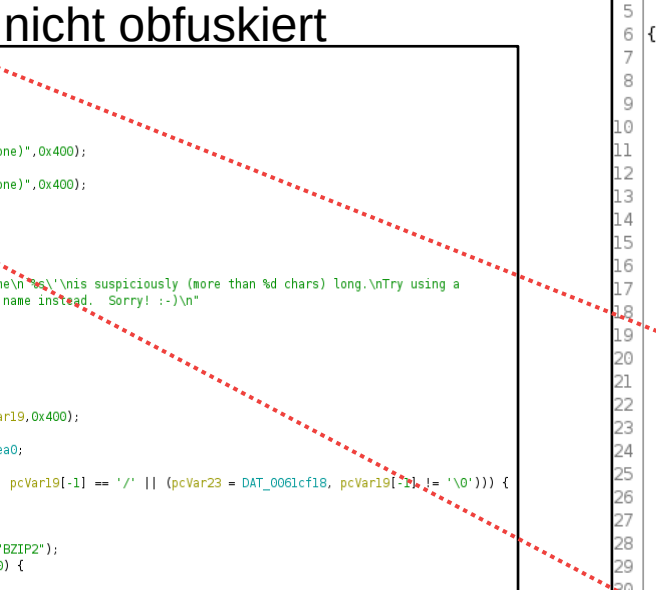
Abbildung 12: Ausschnitt des Funktionsaufrufgraphen der *main*-Funktion des nicht obfuskierten Programms

Der Funktionsaufrufgraph der *main*-Funktion des obfuskierten Programms wiederum kann aufgrund zu vieler ausgehender Referenzen durch Ghidra nicht abgebildet werden. Aus der textuellen Auflistung können zwar die einzelnen Funktionen (ehemals Basic Blocks), die durch *pthread_create* aufgerufen werden, entnommen werden, jedoch ist die Reihenfolge der Abarbeitung der einzelnen Blöcke ohne Kenntnis des Transformationsprozesses nicht rekonstruierbar. Hinzu kommt, dass

durch die zweite erzeugte Aufrufe Ebene nicht die eigentlichen (also ursprünglichen) Unterfunktionen dargestellt werden. Kann ein Programm normalerweise abstrakt durch die Struktur von Funktionsaufrufen verstanden werden, wird diese Praxis durch die ausgedehnte Verschachtelung erheblich behindert.

Als Nächstes soll die Analysierbarkeit des Programms anhand der Ausgabe des Decompilers untersucht werden. Wie bereits festgestellt, sind die aus der *main*-Funktion ableitbaren Informationen recht begrenzt; die *main*-Funktion enthält lediglich die Systemaufrufe unter Nutzung von Speicheradressen, welche für das Verständnis des Programms assoziiert werden müssen. Dementsprechend soll an dieser Stelle beispielhaft die Decompiler Ausgabe einer der neuen Funktionen demonstriert werden (siehe Codeausschnitt 9).

nicht obfuskiert



obfuskiert

```

53  _DAT_0061c6c0 = 0;
54  DAT_0061c6bc = 0x1e;
55  DAT_0061cf0c = 0;
56  DAT_0061c6e4 = 0;
57  signal(0xb,FUN_00402340);
58  signal(7,FUN_00402340);
59  strncpy(&DAT_0061cb00,"(none)",0x400);
60  DAT_0061cf00 = 0;
61  strncpy(&DAT_0061cf0,"(none)",0x400);
62  DAT_0061caf0 = 0;
63  pcVar19 = *param_2;
64  sVar18 = strlen(pcVar19);
65  if (0x400 < sVar18) {
66      fprintf(stderr,
67          "bzip2: file name\n'%s'\nis suspiciously (more than %d chars) long.\nTry using a
68          reasonable file name instead. Sorry! :-)\n"
69          ,pcVar19,0x400);
70      if (DAT_0061c6e4 < 1) {
71          DAT_0061c6e4 = 1;
72      }
73      goto LAB_0040232e;
74  }
75  strncpy(&DAT_0061bea0,pcVar19,0x400);
76  DAT_0061c2a0 = 0;
77  DAT_0061cf18 = &DAT_0061bea0;
78  pcVar19 = &DAT_0061bea1;
79  while ((pcVar23 = pcVar19, pcVar19[-1] == '/' || (pcVar23 = DAT_0061cf18, pcVar19[-1] != '\0'))) {
80      pcVar19 = pcVar19 + 1;
81  }
82  pbVar20 = (byte *)getenv("BZIP2");
83  if (pbVar20 == (byte *)0x0) {
84      __ptr = (char **)0x0;
85  }
86  else {
87      uVar34 = (ulong)*pbVar20;
88      __ptr = (char **)0x0;
89      if (*pbVar20 != 0) {
90          ppuVar21 = __ctype_b_loc();
91          do {

```

```

4  void FUN_0040bc70(void)
5  {
6      size_t sVar1;
7
8      DAT_00627898 = 0;
9      DAT_00627894 = 0;
10     DAT_006278ac = 0;
11     DAT_006280db = 0;
12     DAT_006278b8 = 1;
13     DAT_0062747c = 0;
14     DAT_006278b0 = 9;
15     DAT_00627cca = 0;
16     DAT_006280da = 0;
17     DAT_006278a0 = 0;
18     DAT_00627890 = 0;
19     DAT_006278bc = 0x1e;
20     DAT_006280dc = 0;
21     DAT_006278b4 = 0;
22     signal(0xb,FUN_004049b0);
23     signal(7,FUN_004049b0);
24     strncpy(&DAT_00627cd0,"(none)",0x400);
25     DAT_006280d0 = 0;
26     strncpy(&DAT_006278c0,"(none)",0x400);
27     DAT_00627cc0 = 0;
28     DAT_00625130 = *DAT_00624ee8;
29     sVar1 = strlen(DAT_00625130);
30     if (0x400 < sVar1) {
31         sem_post((sem_t *)&DAT_00625300);
32         return;
33     }
34     sem_post((sem_t *)&DAT_00625360);
35     return;
36 }
37

```

Codeausschnitt 9: Ausgabe des Decompilers zur Anfangsfunktionalität der *main*-Funktion

Es handelt sich hierbei erneut um den Anfang der *main*-Funktion. Mit Hilfe der rot gestrichelten Linie wird die Zuordnung des Ausschnitts des ersten Basic Blocks indiziert. Während auf der linken Seite die Fortsetzung des Ablaufs unmittelbar ersichtlich ist (bei Eintritt der *if*-Bedingung folgt eine Fehlerausgabe), muss auf der rechten Seite zunächst die Referenz des der Funktion *sem_post* übergebenen Speicherbereichs aufgelöst werden. Dies und die allgemein hohe Anzahl an Parallelisierungsprimitiven führt mutmaßlich zu einer Informationsüberlastung, insbesondere durch die große Entfernung der Abhängigkeiten wie bereits im Unterabschnitt „Wirksamkeit“ festgestellt. Je stärker die Verflechtung beziehungsweise Verschachtelung hierbei ist, desto aufwendiger wird es dem Kontrollfluss auf diese Weise zu folgen.

Eine automatisierte statische Analyse ist, abgesehen von heuristischen Methoden, die die hier beschriebenen Anomalien erkennen und als verdächtig einstufen, ohne Kenntnis der Obfuskation nicht wirkungsvoll. Die Möglichkeiten, solch ein obfuskiertes Programm automatisiert zu verarbeiten, sollen im Abschnitt „Ansätze für Gegenmaßnahmen“ aufgezeigt werden.

Abschließend ist anzumerken, dass die durch die Transformation entstehenden Implikationen hier nur beispielhaft an der *main*-Funktion erläutert wurden. Durch die Anwendung der Transformation auf mehrere oder alle Funktionen erhöht sich entsprechend die Komplexität.

4.3.2 Dynamische Analyse

Ein Teil der bei der statischen Analyse erläuterten Schwierigkeiten ist bei der manuellen dynamischen Analyse implizit enthalten, insbesondere die Verteilung eigentlich zusammenhängender Funktionalität einschließlich ihrer Variablen. Bei der Analyse mit einem Debugger kommt die generelle Belastung durch Parallelität hinzu, hierbei ist die außergewöhnlich hohe Anzahl an Threads hervorzuheben. Folglich ist für die Verfolgung des Programmablaufs bei jeder Verzweigung ein umständlicher Wechsel in einen neuen Thread nötig, was die Effizienz beim Reverse Engineering erheblich reduziert.

Ein weiterer Ansatz ist das Setzen eines Breakpoints in einer für die Analyse interessanten Bibliotheks- oder Systemfunktion, zum einen um die Übergabeparameter zu untersuchen und zum anderen um den Weg zu bestimmen, also über welche vorherigen Funktionsaufrufe die Funktion erreicht wurde (genannt Stacktrace). Hierdurch lässt sich ein abstraktes Modell des Programms anhand der Funktionsaufrufe erstellen, ähnlich einem Funktionsaufrufgraphen (siehe Unterabschnitt „Statische Analyse“). Wohingegen Ersteres weiterhin möglich ist, wird Letzteres effektiv verhindert. Als Beispiel kann die Gegenüberstellung in Codeausschnitt 10 betrachtet werden, wo die Stacktrace-Ausgabe von GDB zur *fwrite*-Funktion beim obfuskierten und nicht obfuskierten Beispielpogramm *bzip2* zu sehen ist.

Beim nicht obfuskierten Programm kann der Aufrufpfad vom Programmeinstiegspunkt #5 über die *main*-Funktion #4 und drei weiterer Unterfunktionen #3 - #1 bis zur *fwrite*-Funktion #0 nachvollzogen werden. Beim obfuskierten Programm hingegen ist der Einstiegspunkt #3 die Systemfunktion *clone*, welche bei der Erzeugung eines neuen Threads verwendet wird. Die erste Programmfunktion in diesem Pfad #1 ist eine der bei der Obfuskation neu erzeugten Funktionen, welcher direkt der untersuchte Systemaufruf #0 folgt; der Programmpfad ist folglich nicht ersichtlich.

Folgende absolute Zahlen konnten beim ersten Erreichen der *fwrite*-Funktion gemäß Codeausschnitt 10 festgestellt werden³⁴: Zu diesem Zeitpunkt waren insgesamt 294 Threads aktiv und es wurden bis zu diesem Zeitpunkt insgesamt 9.096.136 Threads erstellt. Dies verdeutlicht die immense Fülle von Threads, die es zu handhaben gilt.

Der Effekt auf die automatische dynamische Analyse mit einer Sandbox spiegelt sich überwiegend bei der Aufzeichnung der Systemfunktionsaufrufe wider. Neben den ursprünglichen Systemaufrufen kommen die Systemaufrufe im Zusammenhang mit der Obfuskation hinzu (überwiegend die Erstellung von Threads mit *NtCreateThread*, aber auch zahlreiche Hilfsfunktionen auf niedriger Abstraktionsstufe wie *NtClose* oder *NtDuplicateObject*). Diese können jedoch gegebenenfalls gefiltert wer-

³⁴ Hierbei wurden alle Funktionen der Dateien *bzip2.c* und *bzlib.c* bei der Transformation miteinbezogen.

den, sofern keine Parallelitätsmechanismen im reinen Programm vorgesehen waren. Andernfalls würde die Vermischung der mit der Obfuskation eingebauten Funktionen mit den im Programm bereits vorhandenen Funktionen mit Bezug zur Parallelisierung die Wirksamkeit steigern.

obfuskiert

```
#0  __GI__IO_fwrite (buf=0x7fff88000fc8, size=1, count=5000,
    fp=0x7fff88000d90) at iofwrite.c:31
#1  0x000000000041e79d in ?? ()
#2  0x00007ffff7bbb6db in start_thread (arg=0x7fff4cf89700) at
    pthread_create.c:463
#3  0x00007ffff78e471f in clone () at
    ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
```

nicht obfuskiert

```
#0  __GI__IO_fwrite (buf=0x61d708, size=1, count=5000, fp=0x61d4d0) at
    iofwrite.c:31
#1  0x0000000000406ea4 in ?? ()
#2  0x0000000000402abe in ?? ()
#3  0x000000000040219a in ?? ()
#4  0x00007ffff7a03bf7 in __libc_start_main (main=0x401490, argc=2,
    argv=0x7fffffffdf08, init=<optimized out>, fini=<optimized out>,
    rtdl_fini=<optimized out>, stack_end=0x7fffffffdef8)
    at ../csu/libc-start.c:310
#5  0x00000000004013ca in ?? ()
```

Codeausschnitt 10: Gegenüberstellung des Stacktraces beim obfuskierten und nicht obfuskierten Programm

4.4 Auswirkungen auf die Malware Detektion

Bei der Untersuchung der Auswirkungen auf die Malware Detektion können eine Vielzahl von Methoden angewandt und zahlreiche Gesichtspunkte begutachtet werden. Im Folgenden soll möglichst ein Bezug zu den im Kapitel „Grundlagen“ erläuterten Aspekten hergestellt werden.

Wie bereits im Eingang des aktuellen Kapitels beschrieben, kann lediglich Quellcode für Prüfungen des Transformationsprozesses herangezogen werden. Sofern Eigenschaften unabhängig von gewissen Voraussetzungen untersuchbar sind, können die Ergebnisse von gutartigen Programmen entsprechend auf Malware übertragen werden. Als Voraussetzungen sei beispielsweise genannt, dass das Programm in einer Signaturdatenbank enthalten sein muss oder tatsächlich maliziöse Aktionen durchführt. Für die hiervon nicht betroffenen Verfahren konnte im Internet eine überschaubare Anzahl von Quellcodes für Malware gefunden werden, die im Eingang des Kapitels vorgestellt wurden. In diesem Zusammenhang ist darauf hinzuweisen, dass aufgrund der geringen Anzahl von Quellcodes die Repräsentativität der Untersuchung nicht gegeben und deshalb als Stichprobe zu betrachten ist.

Für die folgenden Untersuchungen muss sich vergegenwärtigt werden, dass die hier vorgestellte Obfuskation vorrangig die Sektion betrifft, in welcher sich die Maschineninstruktionen befinden (nämlich die *.text-section*). Auch die Datensektionen sind durch die Verlagerung des Datenflusses in den globalen Kontext in einem gewissen Ausmaß hiervon betroffen. Es gibt jedoch auch Bereiche,

die komplett unverändert bleiben. Hervorzuheben sind hierbei eindeutige Zeichenketten, die gewöhnlich eine recht günstige und zuverlässige Wiedererkennung ermöglichen. Auch aus diesem Grund ist die Kombination mit komplementären Obfuskationstechniken notwendig.

Einige Detektionsmethoden kombinieren signaturbasierte, verhaltensbasierte und heuristische Analysetechniken. Zudem existieren Überschneidungen zwischen den Techniken, beispielsweise wenn Sequenzen von Systemaufrufen durch die Beobachtung des Verhaltens bestimmt und anschließend als Signatur verarbeitet werden. Im Folgenden wird die Unterteilung anhand der Komponente vorgenommen, die am stärksten von der hier vorgestellten Obfuskation betroffen ist.

4.4.1 Signaturbasierte Detektion

Hash

Wie bereits im Abschnitt „Malware Detektion“ beschrieben, ist die Nutzung herkömmlicher Hash-Algorithmen wie MD5 oder SHA1 höchstens bei der Wiedererkennung einer bereits im Umlauf befindlichen, konkreten Datei zielführend. Deshalb soll an dieser Stelle nur ein lokal sensitiver Hash betrachtet werden, nämlich der Trend Micro Locality Sensitive Hash (TLSH) [15]. In Codeausschnitt 11 sind einige Lokal Sensitive Hashes von ausführbaren Dateien³⁵ des Beispielsprogramms bzip2 dargestellt. Hierbei wurden auch zwei unterschiedliche Compiler verwendet. Zum einen der Clang Compiler, welcher der offizielle C/C++ Compiler des LLVM Projekts ist, und zum anderen der GCC Compiler [93]. Bei der Obfuskation wurde jede unterstützte Funktion der Transformation unterzogen.

```
1 | A3A35853B2932AFECB6EC5795A5A1626F671706A03187D3F75C9EA303F06A781F04324 Clang, nicht obfuskirt
2 | 2AB450A3A6980CBDD25BD2B7576FC231D024F47022727C057485A0F29FED56A2A346 Clang, obfuskirt
3 | 4AB450A3A6990CB9D2ABD2775B5FD231D014F17022727C097485A0F29FEC5663A306 Clang, obfuskirt, mit Zufall
4 | CC934B49F5A61B7AC486C035061F5B31AB28FD090722377F7848F630BF07AA91F6D669 GCC, nicht obfuskirt
```

Codeausschnitt 11: TLSHs von einigen Samples des bzip2-Programms

Hier ist erkennbar, dass die Anwendung der Obfuskation zu einem deutlich anderen TLSH führt (vergleiche Hash 1 und 2). Die Distanz zwischen den beiden TLSHs beträgt 426, weshalb anhand dieses Hashes auf keinen Zusammenhang zwischen beiden Dateien zu schließen wäre (vergleiche beispielsweise die Distanz zwischen den Programmen xz und zip von 252³⁶). Eine Distanz von 129 lässt sich jedoch bereits durch die Nutzung eines anderen Compilers erreichen (vergleiche Zeile 1 und 4). In Zeile 3 wurde die Reihenfolge, in der die Funktionen behandelt und in der die Basic Blocks verlagert werden, randomisiert, um Eigenschaften eines polymorphen Generators zu realisieren. Zu sehen ist, dass dies zu Unterschieden beim TLSH führt (die sich unterscheidenden Ziffern sind farblich markiert). Zwar ist die in diesem Fall bestimmbare Distanz von 17 keineswegs maßgeblich, kann aber trotzdem als Beitrag eines umfangreicheren Wandlungsprozesses betrachtet werden.

³⁵ ELF, 64-bit, stripped, Optimierungsstufe 2

³⁶ xz (Version 5.2.2) und zip (Version 3.0) einer Ubuntu bionic Distribution

YARA

Ähnlich den Datenbanken für die Erkennung von Malware mit Hashes existieren ebenfalls Repositories mit YARA-Regeln. Aufgrund des musterbasierten Detektionsmechanismus ist es mit dieser Methode grundsätzlich möglich, Malware Varianten zu detektieren beziehungsweise detektierte Malware einer bekannten Familie zuzuordnen [17]. Doch auch die Erkennung neuartiger Malware anhand typischer Muster stellt eine Anwendungsmöglichkeit dar. Bei der Anwendung der Regeln aus den öffentlichen Repositories [94] auf die hier generierten Samples musste jedoch festgestellt werden, dass nahezu keine Samples (weder nicht obfuskiert noch obfuskiert) als explizit maliziös eingestuft wurden³⁷. Die wenigen Detektionen waren zudem auf konstante Zeichenketten außerhalb der Codesektion zurückzuführen, welche durch Kombination mit einer anderen Obfuskationstechnik zu behandeln wären. Folglich bietet dieser Untersuchungsansatz keine Grundlage für eine aussagekräftige Beurteilung der hier betrachteten Detektionsmethode.

Als alternativer Ansatz wurden dankenswerterweise durch die Mitarbeiter des Fraunhofer FKIE individuelle YARA-Regeln auf Basis der für diese Arbeit kompilierten, nicht obfuskierten Samples mit Hilfe des YARA-Signators generiert. Beim YARA-Signator erfolgt die Generierung der Regeln anhand von ausgewählten Instruktionsfolgen, die für eine bestimmte Malware Familie besonders relevant sind und nicht in anderen Malware Familien auftreten. Letzteres dient der Vermeidung von falschen Detektionen, die insbesondere bei Nutzung von Code aus Bibliotheken durch die jeweiligen Samples entstehen können. Der Fokus auf die Codesektion impliziert, dass die weniger stark von der Thread-basierten Obfuskation betroffenen Sektionen keinen Einfluss auf das Untersuchungsergebnis nehmen. Die Generierung der Regeln erfolgte unter Einbeziehung des Datenbestands von Malpedia. Die Unterteilung in Malware Familien erfolgte anhand des zugehörigen Projekts. [95]

In Tabelle 3 sind die Detektionen für die 64-bit und 32-bit Varianten in jeweils nicht obfuskiertem Form und obfuskiertem Form aufgeführt. Bei der nicht obfuskierten Form wurde die mit dem Clang Compiler erzeugte Variante gewählt. Wie an den Werten zur Gesamtzahl der Detektionen zu erkennen ist, reduziert sich die Detektionsrate nach Anwendung der Obfuskation signifikant. Dies ist vorrangig auf die veränderte Anordnung der einzelnen Basic Blocks zurückzuführen, wodurch die durch den YARA-Signator extrahierten Instruktionsfolgen voneinander getrennt wurden. Doch auch andere Ursachen sind denkbar. Zum einen besteht die Möglichkeit, dass durch die in der Zwischensprache durchgeführten Änderungen zusätzliche Unterschiede beim final emittierten Maschinencode entstehen. Zum anderen können auch nicht auf die Thread-basierte Obfuskation zutreffende Annahmen des YARA-Signators bei der Generierung der Regeln zu dieser starken Abnahme der Detektionen führen. So wird als Bedingung für die Detektion stets eine maximale Dateigröße angegeben. Wird diese Bedingung entfernt, so steigt die Anzahl für die 64-bit Variante um zwei und für die 32-bit Variante um sechs Detektionen.

Letztlich ist auf die Einschränkung dieses speziellen Versuchsaufbaus zu achten, dass teilweise Samples einer Familie zugeordnet wurden, die trotzdem im Projekt nicht notwendigerweise den Quellcode teilen. Zudem ist zu beachten, dass in der Praxis nach der Begegnung mit einem mit der Thread-basierten Obfuskation transformierten Sample dieses in den Prozess der Signaturgenerierung einfließen würde, um gegebenenfalls polymorphe Varianten zu detektieren, die durch Para-

³⁷ Der Grund hierfür ist ohne umfassende Untersuchung nicht ersichtlich.

metrisierung der Obfuskation erstellt wurden. Insbesondere bei der Wahl einer kleineren Einteilungsgröße ist von einer Steigerung der Effektivität der Obfuskation auszugehen.

Programmname	64-bit		32-bit	
	n. obfus.	obfus.	n. obfus.	obfus.
Alina / alina.exe	X	O	X	O
AryanRAT / AryanRAT.dll	X	O	X	O
BASHLITE / client	X	O	X	O
BASHLITE / server	X	O	X	O
Carberp / anti_rapport.exe	-	-	X	O
Carberp / BinToHex.exe	X	O	X	O
Carberp / BootkitInstallReport.exe	-	-	X	O
Carberp / BootkitRunBot.dll	-	-	X	O
Carberp / BotLoader2.exe	-	-	X	O
Carberp / CoreDll.dll	-	-	X	O
Carberp / ddos.dll	-	-	X	X
Carberp / Full.exe	-	-	X	O
Cyberbot / CyberBot.exe	-	-	X	O
Dexter / Dexter.exe	X	O	X	O
Dokan / dokan_mount.exe	X	X	X	X
Dokan / dokan.dll	X	X	X	O
Grum / Grum.exe	-	-	X	O
KAAL_BHAIRAV / KAAL_BHAIRAV	X	O	X	O
Lightaidra / lightaidra	X	O	X	O
LiquidBot / LiquidBot.exe	-	-	X	O
Meterpreter / elevator.dll	X	O	X	O
Meterpreter / screenshot.dll	-	-	X	X
Mettle / mettle	X	X	X	X
Mimikatz / mimikatz.exe	X	O	X	O
Mimikatz / mimilib.dll	X	X	X	X
Mimikatz / mimilove.exe	X	X	X	X
Mimikatz / mimispool.dll	X	X	X	X
Mirai IoT Botnet / bot	X	O	X	O
Mirai IoT Botnet / loader	X	O	X	O
MyDoom.A / bin2c.exe	X	O	X	O
MyDoom.A / cleanpe.exe	X	O	X	O
MyDoom.A / client.exe	X	O	X	O
MyDoom.A / cryptX.exe	X	O	X	O
MyDoom.A / MyDoomA.exe	X	O	X	O
MyDoom.A / rotX3.exe	X	O	X	O
MyDoom.A / xproxy.dll	X	O	X	O
ogwOrm / ogwOrm.exe	X	X	X	X
pnsan / pnsan	X	O	X	O
Polymorphic Virus / encrypt.exe	X	O	X	O
rBot / rBot.exe	-	-	X	O
ReflectiveDllInjection / Inject.exe	X	O	X	O
ReflectiveDllInjection / ReflectiveLoader.dll	X	O	X	O
VXHeaven / Exploit.Linux.Interbase	O	O	O	O
VXHeaven / Exploit.Linux.Nhttpd	X	O	X	O
VXHeaven / Net-Worm.Linux.Slapper	X	O	X	O
VXHeaven / Virus.Unix.Adrastea	X	O	X	O
VXHeaven / Worm.FreeBSD.Block	X	O	X	O
XOR-USB / xor-usb.exe	X	O	X	O
XOR-USB / xor.exe	X	O	X	O
xTBot / xTBot.exe	-	-	X	O
Gesamt	36	7	49	8

Legende:

X = detektiert

O = nicht detektiert

- = nicht verfügbar

Tabelle 3: Detektionen durch die vom YARA-Signator generierten YARA-Regeln

Andere Signaturgrundlagen

Im Weiteren soll ein Bezug zu einigen wissenschaftlichen Publikationen hergestellt werden, welche die Erkennung von neuer Malware oder gewandelter Malware mit Hilfe von Signaturgenerierung auf Basis anderer Eigenschaften untersucht.

In einer Vielzahl (auch aktueller) Publikationen werden Kontrollflussgraphen als Basis für die Detektion beziehungsweise Generierung von Signaturen verwendet [18, 19, 20]. Ihnen allen ist, wie in [18] beschrieben, gemeinsam, dass die Effektivität dieser Strategie auf dem Grad der Komplexität der Kontrollflussgraphen beruht. Dieser Prämisse folgt, dass diese Vorgehensweise bei einem mit der Thread-basierten Obfuskation transformierten Programm wirkungslos ist, da dem Kontrollflussgraphen jegliche Struktur und somit Komplexität entnommen wird.

In [22] werden über Basic Blocks sogenannte Feature Hashes für die Detektion von Malware verwendet. Diese Methode ist uneingeschränkt anwendbar, falls die Einteilungsgröße bei der Verlagerung größer oder gleich einem Basic Block ist. Wird die Einteilungsgröße kleiner gewählt, so verliert die Methode an Wirkung, sofern die in dieser Publikation beschriebene Codenormalisierung nicht eine geeignete Deobfuskation umfasst. Sofern kleinere Bereiche als Basic Blocks als Signatur genutzt werden (wie in [96] und [73]), ist ebenfalls ein Zusammenhang zwischen der Effektivität der Obfuskation und der Verkleinerung der Einteilungsgröße der zu verlagernden Funktionalität zu erwarten, da die Instruktionen in den Bereichen folglich voneinander getrennt werden.

Auch die statische Analyse der Systemaufrufe oder Aufrufe bekannter Bibliotheken eines Programms wird ohne Kenntnis der Obfuskationsmethode erschwert, sofern wie bei [33] der Kontrollfluss in die Generierung von Signaturen einfließt und davon ausgegangen wird, dass der Kontrollfluss nach dem Disassemblieren ohne Weiteres bestimmt werden kann.

4.4.2 Verhaltensbasierte Detektion

Allgemein

Für die Analyse des Verhaltens von Software wird gewöhnlich die Software in einer isolierten und umfassend überwachten Umgebung ausgeführt. Einige Antivirus-Hersteller bieten hierfür kommerzielle Lösungen an; in dieser Arbeit soll hierfür die open-source Cuckoo Sandbox auf obfuskierte Programme angewandt werden.

Als Prüfumgebung wurde die Cuckoo Sandbox (Version 2.0.7) in Verbindung mit Virtual Box (Version 6.1) [97] eingerichtet. Als virtuelle Maschine wurde das Windows 7 Betriebssystem gewählt, da Windows als Ziel von Viren beliebter als die Alternative Linux ist (siehe auch Abschnitt „Malware Detektion“), das Windows 7 OS offiziell von Cuckoo empfohlen wird und bei ersten Untersuchungen für diese Arbeit mit dem Windows 10 OS die bemerkbare Fehleranfälligkeit die Empfehlung bestätigt hat. Überdies musste auch bei Windows 7 festgestellt werden, dass ausführbare Dateien für die 32-bit Architektur (unabhängig von der Architektur des Betriebssystems) zu bevorzugen sind. Dies geht aus der Feststellung hervor, dass bei einem trivialen Sample für die 64-bit Architektur die Ausgabe auf der Konsole nicht erkannt wurde. Dieser Umstand mag darauf zurückzuführen sein, dass Viren primär für die 32-bit Architektur kompiliert werden, da diese auch auf einem 64-bit System ausführbar sind, der umgekehrte Fall aber nicht möglich ist, und deshalb der Fokus bei der Entwicklung der Cuckoo Sandbox stärker auf 32-bit Samples liegt.

Im Allgemeinen wurde die Mehrheit der Samples mit einer geringen Punktzahl zwischen 0 bis 1 von 10 als nicht verdächtig bewertet. Dies ist auf eine Vielzahl von Faktoren zurückzuführen. Zunächst ist die Funktionsweise der genutzten Programme nicht vollumfänglich bekannt. Folglich können fehlende Argumente oder eine falsche Konfiguration während der Kompilierung dazu führen, dass die maliziösen Aktivitäten nicht durchgeführt werden. Teilweise werden nur einzelne verdächtige Aktionen durchgeführt, sodass die summierte Punktzahl niedrig bleibt. Aber auch weitere Abhängigkeiten sind denkbar. Bei einem Sample wurde durch die Analysesoftware eine Prüfung der Größe des Arbeitsspeichers detektiert, was eine Maßnahme zur Erkennung einer Sandbox sein kann, sodass die weitere Ausführung von verdächtigem Verhalten eingestellt würde. Letztlich kann auch ein Fehler bei der Obfuskation nicht ausgeschlossen werden, welcher zu einem nicht funktionsfähigem Sample geführt hat. So wurde bei manueller Prüfung des obfuskierten 32-bit Programms Grum festgestellt, dass die Arbeitsspeicherlimitierung des Prozesses erreicht wurde und infolgedessen die Aktivität eingestellt wurde.

Eine besondere Rolle nehmen dynamische Bibliotheken ein: Ohne angepasste Analyseinstellungen bei Cuckoo wird lediglich mit Hilfe des *rundll32* Kommandos die *DllMain*-Funktion ausgeführt. Dementsprechend besteht die Möglichkeit, dass andere exportierte Funktionen mit den bösartigen Routinen, die gegebenenfalls in Kombination mit anderen Techniken³⁸ zum Einsatz kommen, nicht bemerkt werden. Zudem wurde beobachtet, dass vom *rundll32* Kommando getätigte Systemaufrufe von Cuckoo teilweise als verdächtig eingestuft wurden und bei der Bewertung des Samples miteinbezogen wurden.

In diesem Zusammenhang konnten auch weitere Verhaltensweise der Analysesoftware, die nicht den Erwartungen entsprachen, festgestellt werden: Die in zahlreichen (auch gutartigen) Samples vorhandene *._RDATA-section* wurde als verdächtig eingestuft; die Installation von ausführbaren Dateien in *C:\Windows* wurde teilweise nicht als verdächtig eingestuft; der typisch wiederholte Aufruf von *Process32Next* zur Suche nach einem Prozess wurde teilweise nicht erkannt; die stichprobenartige Wiederholung von Untersuchungen führte stellenweise zu (minimalen) Abweichungen bei den Ergebnissen; CyberBot und rBot haben sich bis zum Ende der Analyselaufzeit achtmal selbst neu gestartet, die *explorer.exe* Datei geöffnet und es wurde die Anzahl der Prozessoren geprüft, trotzdem wurden beide mit der niedrigsten Punktzahl als gutartig bewertet. Gemäß dieser Erkenntnisse muss von einer bedingten Robustheit der Analysesoftware ausgegangen werden, welche sich auf die Ergebnisse der durchgeführten Untersuchung auswirken kann.

Für die Detektion von Netzwerkaktivität wurde mit INetSim [98] der Internetzugriff simuliert, es wurde allerdings kein Netzwerkzugriff aufgezeichnet. Die im Unterabschnitt „Detektion durch kommerzielle Scanner“ vorgestellten Ergebnissen korrelieren in diesem Zusammenhang mit den hiesigen, lediglich beim Grum Programm wurde abweichend ein Zugriff festgestellt. Diese Information war jedoch erst zu einem späteren Zeitpunkt nach offiziellem Abschluss der Analyse verfügbar, weshalb hierbei von einer längerfristigen Untersuchung ausgegangen wird, gegebenenfalls sogar mit Systemneustart. Im Gegensatz hierzu beträgt die Standardlaufzeit bei Cuckoo zwei Minuten, sodass durch eine entsprechende Verzögerung dieses Verhalten nicht aufgezeichnet werden konnte, obwohl eine Verzögerung direkt nach Prozessstart von der Analysesoftware erkannt und die Laufzeit entsprechend verlängert wurde.

³⁸ Es besteht beispielsweise die Möglichkeit, dass durch die *DllMain* ein Autostarteintrag generiert wird, durch welchen eine exportierte Funktion der Bibliothek bei Systemneustart ausgeführt wird.

Da eine Betrachtung jeglicher Unterschiede der einzelnen Varianten des Programme des vorliegenden Korpus aufgrund des Umfangs und der teilweise starken Ähnlichkeit nicht zielführend ist, sollen an dieser Stelle die herausragenden Ergebnisse zum Programm Alina aus dem TheZoo-Repository ausführlicher vorgestellt werden. Denn bei diesem Programm steht die Punktzahl in Höhe von 5,6³⁹ der nicht obfuskierten Variante einer Punktzahl von 0,4⁴⁰ der obfuskierten Variante gegenüber. Die Möglichkeit eines fehlerhaften Samples wurde ausgeschlossen, indem die Existenz der bei der nicht obfuskierten Variante entstandenen und detektierten Artefakte nach manueller Ausführung der obfuskierten Variante geprüft wurde. Es wurde lediglich eine deutliche Verzögerung bis zum Eintritt der erwarteten Aktivitäten im Gegensatz zur nicht obfuskierten Variante festgestellt.

Bei der obfuskierten Variante wurde lediglich ein statisches Merkmal (*_RDATA-section*) sowie ein Systemaufruf zur Bestimmung des Computernamens als verdächtig eingestuft. Bei der nicht obfuskierten Variante hingegen wurden hinzukommend neun weitere verdächtige Aktivitäten aufgezeichnet, die von der Platzierung einer ausführbaren Datei über mehrere Einzelaktivitäten zur Injektion in einen anderen Prozess bis zur Erkennung von erstellten Objekten reichen, die bereits öffentlich bekannt sind und der Malware Familie eindeutig zugeordnet werden können. Wie bereits im Unterabschnitt „Dynamische Analyse“ dargestellt, entsteht durch die Obfuskation eine hohe Anzahl von zusätzlichen Systemaufrufen, in diesem Fall in Höhe von 262.625 (davon 65.498 *CreateThread*, 65.420 *NtDuplicateObject* und 130834 *NtClose*) im Vergleich zur nicht obfuskierten Variante mit nur 509 Systemaufrufen.

Der Grund für diesen fundamentalen Unterschied bei der Bewertung der Samples konnte im Rahmen dieser Arbeit nicht abschließend geklärt werden. Da bei beiden Varianten die eingestellte Höchstlaufzeit ausgereizt wurde, ist eine mögliche Erklärung, dass durch die Obfuskation eine entsprechend hohe Verzögerung entstanden ist, sodass die maliziösen Aktivitäten nicht durchgeführt wurden. Dem steht jedoch entgegen, dass bei der manuellen Prüfung die ersten Artefakte bereits innerhalb der ersten Minute auffindbar waren. Die Bestimmung weiterer möglicher Ursachen ist nicht trivial, hierfür müsste die Logik der Analysesoftware eingehender untersucht werden. Dennoch ist der Einsatz der Obfuskation zur Verzögerung der Ausführung denkbar. In diesem Zusammenhang ist die durch den Performanzverlust zusätzlich entstehenden Prozessorlast zu berücksichtigen. Es konnte vorwiegend eine Gesamtauslastung von etwas mehr als 50 % bei zwei Prozessoren beobachtet werden. Bei steigender Prozessoranzahl fällt dieser Wert entsprechend geringer und unauffälliger aus. Diese Kosten wären individuell nach potentielltem Ziel einzukalkulieren.

Letztlich wurde durch die Analysesoftware nur einmal eine Sequenz von Systemaufrufen detektiert, sonst wurden die Systemaufrufe einzeln als Indikator herangezogen. Und selbst diese Sequenz mit dem *Process32Next* Systemaufruf wurde nicht zuverlässig erkannt (wie bereits oben angedeutet). Dementsprechend konnte die Wirksamkeit der durch die Obfuskation eingebrachten Systemaufrufe im Bezug auf die Erkennung von Sequenzsignaturen nicht verifiziert werden.

Im Folgenden werden spezialisierte Methoden untersucht, die teilweise für die Gewinnung der Informationen auch die hier beschriebene Cuckoo Sandbox verwenden [99], sodass die hier gewonnenen Erkenntnisse ebenfalls bei den betroffenen Methoden zu erwarten sind.

39 bei Wiederholung der Analyse sogar 7,4

40 bei Wiederholung der Analyse identisch

Spezialisierte Methoden

Auch hier ist ein Bezug zu wissenschaftlichen Publikationen vorgesehen, die einen konzeptionellen Beweis zur Detektion von Malware durch Analyse des Verhaltens beschreiben. Die Methoden in diesem Unterabschnitt sind nicht öffentlich verfügbar, weshalb die Reproduzierbarkeit im Rahmen dieser Arbeit nicht möglich ist und keine praktischen Prüfungen durchgeführt werden können.

Eine beliebte Methode zur Abstraktion des Verhaltens der zu analysierenden Software ist die Aufzeichnung der Systemaufrufe [27]. Bei einigen dieser Verfahrensweisen werden zusätzlich die Argumente sowie die Rückgabewerte der Systemaufrufe betrachtet [28]. In [99] wird eine Signatur sogar auf Basis der Häufigkeit von Systemaufrufen generiert. Bei diesen Methoden kann ebenfalls mit einer gewissen Störung der Leistungsfähigkeit hinsichtlich der hinzugekommenen Systemaufrufe der Obfuskation gerechnet werden. Wird wie bei [21] der Kontrollflussgraph beim Verfahren mit einbezogen, so kann wie bei der signaturbasierten Detektion davon ausgegangen werden, dass diese Detektionsform deutlich an Wirksamkeit verliert.

Die bei [100] beschriebene Methode benutzt einen Debugger um die tatsächlich eingesetzten Maschineninstruktionen während der Ausführung zu bestimmen. In dieser Publikation wird zwar nicht auf die Schwierigkeit bei Parallelität eingegangen, sofern der Debugger jedoch geeignet konfiguriert ist und der Wechsel zwischen Threads entsprechend implementiert ist, kann diese Art der Detektion weiterhin zielführend anwendbar sein. Wie bei den hinzugekommenen Systemaufrufen müssten hierbei ebenfalls die hinzugekommenen Instruktionen gefiltert werden.

4.4.3 Heuristische Detektion

Im Zusammenhang mit der heuristischen Detektion ist auch die Erkennung von Obfuskation zu betrachten. Diesbezüglich ist festzustellen, dass die ungewöhnliche Programmstruktur nach Anwendung der Obfuskation in Verbindung mit der hohen Anzahl an Threads durch einige Detektionsmethoden vermutlich als höchst verdächtig eingestuft wird, auch wenn hierzu keine konkrete Beschreibung einer Detektionsmethode gefunden werden konnte, bei der ein solches Verhalten zu erwarten wäre. Hierbei ist jedoch auch zu beachten, dass heuristische Methoden im Allgemeinen in hohem Maße von falschen Alarmen betroffen sind [101]. Zudem ist auch eine hohe Anzahl von fehlenden Alarmen denkbar. Aufgrund dieser neuralgischen Eigenschaft werden diese Methoden (falls überhaupt) vorrangig als Vorstufentest beziehungsweise begleitend eingesetzt [31], sodass zum aktuellen Zeitpunkt von keiner ausgeprägten Erkennungsrate hinsichtlich der Thread-basierten Obfuskation bei auf Produktivsystemen eingesetzten Detektoren auszugehen ist.

Zu den heuristischen Methoden sind unter anderem vergleichs- beziehungsweise ähnlichkeitsbasierte Methoden zu zählen. Beispielsweise werden für die Untersuchung auf Dateien die Kosinus-Ähnlichkeit [72] angewandt oder die Entropie [102] berechnet. Auch der TLSH kann hierfür eingesetzt werden [16]; so bieten die Ergebnisse des Unterabschnitts „Signaturbasierte Detektion“ einen ersten Eindruck, wie wirksam die Methoden hinsichtlich der Obfuskation womöglich sind. In [11] wird Codenormalisierung vor einem Vergleich angewandt. Aufgrund der stark invasiven Obfuskation wäre die Normalisierung limitiert und eine gezielte Deobfuskation notwendig. Ebenfalls basiert der Vergleich wiederum auf strukturellen Eigenschaften eines Programms, welche durch die Obfuskation verschleiert werden.

Detektion mit Hilfe von maschinellem Lernen ist ein recht neues und besonders breites Forschungsfeld, da verschiedene Techniken des maschinellen Lernens auf unterschiedliche Aspekte einer zu analysierenden Probe angewandt werden können. Als Grundlage des Verfahrens werden Opcode-Sequenzen [34], Systemaufruf-Sequenzen [27, 28], die ganze Datei an sich ohne Eigenschaftsextraktion [26] oder sogar ganze Report-Dateien eines Analyseservices [29] verwendet. Als Lernmethode werden künstliche neuronale Netze [26], Entscheidungsbäume und Support Vector Machines (SVM) [74] verwendet. Zudem ist zu beachten, dass die Trainingsdaten für die Algorithmen im Bezug auf maschinelles Lernen meist als Ergebnis einer statischen oder dynamischen Analyse hervor gehen [30] und durch die sich ergebende Abhängigkeit bereits in den vorangegangenen Abschnitten behandelt wurde.

Zwar steht das in [26] vorgestellte Projekt auf Github zur Verfügung [103], jedoch schreiben Raff et al., dass für das Training mit 400.000 Samples und acht verfügbaren GPUs eine Woche benötigt wurde. Diese Ressourcen standen bei der Erstellung dieser Arbeit nicht zur Verfügung, sodass keine praktischen Tests durchgeführt werden konnten. Für das in [27] vorgestellte Projekt Malheur werden wiederum Aufzeichnungen des Verhaltens von Programmen im CWSandbox XML oder Malware Instruction Set (MIST) Format benötigt. Da in dieser Arbeit jedoch die Cuckoo Sandbox zur verhaltensbasierten Analyse verwendet wurde und das MIST Format als veraltet gilt [104], sodass kein funktionierendes Projekt zum Konvertieren der Aufzeichnungen von Cuckoo ins MIST Format auffindbar war, konnte auch dieser Ansatz nicht im Zusammenhang mit der Thread-basierten Obfuskation praktisch untersucht werden.

4.4.4 Detektion durch kommerzielle Scanner

Da keine genauen Informationen über die Funktionsweise kommerzieller Scanner verfügbar ist, weil dies ein Geschäftsgeheimnis ist, um sich gegenüber Konkurrenten herauszustellen, aber auch um für Gegenspieler die Angriffsfläche möglichst gering zu halten, gestaltet sich die Prüfung der Reaktion von kommerziellen Scannern als Black-Box-Verfahren. Zudem ist zu beachten, dass, im Gegensatz zu den vorangegangenen Unterabschnitten, mit hoher Wahrscheinlichkeit mehrere Detektionsmethoden kombiniert werden. Durch die verschiedenen eingesetzten Detektionsmethoden bei den einzelnen Scannern können die Detektionsraten starken Schwankungen unterliegen.

Für die Untersuchung der Auswirkungen auf kommerzielle Scanner ist eine gewisse Anzahl von verschiedenen Antivirus-Produkten erforderlich, da eine stichprobenartige Untersuchung mit zum Beispiel nur Windows Defender und ClamAV aufgrund der geringen Aussagekraft nicht zielführend ist. Um eine geeignete Laborumgebung für diese Zwecke zu erschaffen sind nicht nur umfangreiche Hardwareressourcen notwendig, auch die finanziellen Aufwendungen für die Beschaffung der Scanner ist nicht vernachlässigbar. Aus diesem Grund soll auf den beliebten Online-Dienst VirusTotal [105] ausgewichen werden, welcher auch häufig bei Untersuchungen in anderen Publikationen eingesetzt wurde [9, 26, 106, 107, 108, 109]. Als Nachteil ist hierbei jedoch zu erwähnen, dass nicht bekannt ist, wie die eingesetzten Scanner konfiguriert sind. Insbesondere hinsichtlich Cloud-basierter Unterstützung sind Unterschiede bei der Detektionsfähigkeit denkbar.

In Tabelle 4 ist die Abnahme der Detektionsanzahl durch die Scanner von VirusTotal ersichtlich, die kompletten Messdaten könne im Anhang 7.8 Messdaten zur Detektion durch kommerzielle Scanner eingesehen werden. Die Generierung der Daten erfolgte am 28. Juli und 29. Juli 2021. Hierbei wird

zwischen der 64-bit und 32-bit Variante unterschieden. Zusätzlich wurde geprüft, wie effektiv die Obfuskationsmethode hinsichtlich der Malware Evasion ist, wenn nur die Main-Methode obfuskiert wird („main obfus.“). Aufgrund der Vielfältigkeit dieses Ergebnisses wäre beinahe eine Einzelfallbetrachtung notwendig, dennoch sollen im folgenden die wichtigsten Erkenntnisse geschildert werden. Da für einige Projekte keine 64-bit Samples kompiliert werden konnten und die Detektionsrate bei den 32-bit Samples im Allgemeinen höher ausfiel⁴¹, liegt im Folgenden der Fokus auf den 32-bit Varianten.

Hinsichtlich der Wirksamkeit kann überwiegend eine positive Entwicklung bei der Detektionsrate verzeichnet werden. In 30 von 49 Fällen wurde die Detektionsrate um mindestens eine Detektion verringert, bis zu 16 Detektionen weniger war möglich. Im Mittel konnte mit der Komplettobfuskation die Detektionsrate um circa zwei Detektionen verringert werden, mit der Obfuskation der *main*-Funktion war es im Schnitt eine halbe Detektion.⁴² Werden diese Zahlen unter Berücksichtigung der Möglichkeit zur signaturbasierten Detektion in den konstanten Datensektionen betrachtet, handelt es sich hierbei im Kontext der Malware Evasion um ein nicht vernachlässigbares Ergebnis.

Jedoch muss auch festgestellt werden, dass sich in zehn Fällen die Detektionsrate erhöht hat, in einem Extremfall um zwölf Detektionen. Zudem fällt auf, dass diese starke Tendenz bei der Zunahme nicht bei der jeweiligen anderen Variante ersichtlich ist. Der Grund hierfür ist aufgrund den Black-Box-Eigenschaften dieser Untersuchung nicht ersichtlich, grundsätzlich ist jedoch die Auffälligkeit der Obfuskation nicht zu vernachlässigen. Insbesondere in Verbindung mit Künstlicher Intelligenz können hierbei unerwartete Ergebnisse entstehen. Dies wird bei der Betrachtung der gutartigen Referenzsamples unterstrichen, bei denen eine Detektion durch zwanzig Scanner bei der nicht obfuskierten Variante erfolgte beziehungsweise die Detektionsrate lediglich bei der *main*-obfuskierten Variante um elf Detektionen anstieg.

Bezüglich der konkreten Scanner, die maßgeblich an der Bildung der Zahlen beteiligt sind, lässt sich zunächst kein Muster erkennen. Mehrfach ist sogar der Fall anzutreffen, dass bei obfuskierten und nicht obfuskierten Variante unterschiedliche Scanner das Sample als maliziös eingestuft haben, die Gesamtanzahl jedoch gleich geblieben ist. Vor diesem Hintergrund kann die Obfuskationsmethode besonders bei Kenntnis über das Ziel erfolgreich sein. Ebenfalls konnten Fälle beobachtet werden, die nicht den Erwartungen entsprachen. So wurde beispielsweise die nicht obfuskierte Variante des Programms Alina nicht durch den für die Analyse mit Signaturen bekannten Scanner ClamAV detektiert, die obfuskierte Variante indessen schon.

Abschließend wird angemerkt, dass nach einem erneuten Abruf der Daten am Folgetag festgestellt wurde, dass in einigen Fällen die Detektionsrate angestiegen ist (sogar bei den gutartigen Referenzprogrammen). Die Gründe hierfür sind ebenso unergründbar wie bei den zwölf zusätzlichen Detektionen bei der obfuskierten Variante. Ebenso diffus ist das Bild hinsichtlich der betroffenen Varianten. Zudem konnte beobachtet werden, dass einige Scanner am Folgetag keine Detektion meldeten, die das Samples zuvor als bösartig eingestuft haben. Indizien für eine Weiterverarbeitung der hochgeladenen Samples konnte jedoch bereits auch in anderen Untersuchungen festgestellt werden [107].

41 Dies kann auf den Umstand zurückzuführen sein, dass Viren primär für die 32-bit Architektur kompiliert werden, wie bereits im Unterabschnitt „Verhaltensbasierte Detektion“ angemerkt wurde.

42 Bei den 64-bit Varianten liegen die Werte leicht darunter.

Programmname	Abnahme (64-bit)		Abnahme (32-bit)	
	obfus.	main obfus.	obfus.	main obfus.
Linux				
Lightaidra / lightaidra	1	0	7	9
BASHLITE / client	3	1	4	0
Mirai IoT Botnet / bot	3	4	3	3
Mettle / mettle	0	0	1	0
VXHeaven / Worm.FreeBSD.Block	0	0	1	1
VXHeaven / Exploit.Linux.Nhttpd	1	0	1	0
pnsan / pnsan	3	-1	1	-1
Mirai IoT Botnet / loader	-1	0	0	0
BASHLITE / server	0	1	0	0
VXHeaven / Virus.Unix.Adrastea	1	0	0	0
Windows				
Carberp / Full.exe	-	-	16	2
Dexter / Dexter.exe	6	0	13	-1
Carberp / BotLoader2.exe	-	-	11	0
AryanRAT / AryanRAT.dll	1	0	9	8
Mimikatz / mimikatz.exe	1	0	9	2
LiquidBot / LiquidBot.exe	-	-	8	0
CyberBot / CyberBot.exe	-	-	6	0
rBot / rBot.exe	-	-	6	1
xTBot / xTBot.exe	-	-	5	-1
X0R-USB / xor-usb.exe	0	1	4	0
MyDoom.A / xproxy.dll	9	1	4	0
Carberp / CoreDll.dll	-	-	3	0
Carberp / anti_rapport.exe	-	-	2	-1
Carberp / BootkitRunBot.dll	-	-	2	0
Carberp / ddos.dll	-	-	2	-1
Grum / Grum.exe	-	-	2	2
Mimikatz / mimilove.exe	0	-1	2	3
ogw0rm / ogw0rm.exe	2	0	2	0
Meterpreter / screenshot.dll	-	-	2	2
Alina / alina.exe	3	-1	1	2
Carberp / BootkitInstallReport.exe	-	-	1	0
MyDoom.A / client.exe	0	0	1	1
ReflectiveDllInjection / ReflectiveLoader.dll	1	1	1	0
MyDoom.A / crypt1.exe	-1	-1	0	0
Dokan / dokan.dll	1	1	0	0
Meterpreter / elevator.dll	-2	-1	0	0
MyDoom.A / MyDoomA.exe	10	1	0	2
Carberp / BinToHex.exe	0	0	-1	-2
MyDoom.A / cleanpe.exe	1	1	-1	-1
Polymorphic Virus / encrypt.exe	0	1	-1	-1
MyDoom.A / bin2c.exe	-1	-1	-2	-2
ReflectiveDllInjection / Inject.exe	0	1	-2	0
Mimikatz / mimilib.dll	2	1	-2	-2
Mimikatz / mimispool.dll	1	0	-2	1
MyDoom.A / rot13.exe	1	0	-2	0
X0R-USB / xor.exe	0	0	-4	-2
Dokant / dokan_mount.exe	-1	-1	-12	0
Meterpreter / metsrv.dll	-	3	-	1
<i>Referenz (Windows⁴³)</i>				
BBP Formel	0	1	16	0
bzip2	0	-11	0	0

Tabelle 4: Anzahl der Detektionen durch VirusTotal

43 Bei Linux wurden bei jeglichen Varianten der Referenzprogramme keine Detektion gemeldet.

5 Diskussion

5.1 Vergleich mit ähnlichen Publikationen

Dieser Abschnitt der Arbeit wird bewusst hinter der Beschreibung des Entwurfs und der Evaluation behandelt, damit zum einen für den Leser der Kontext der Thread-basierten Obfuskation hergestellt wurde und zum anderen um die Ergebnisse aus dem Kapitel „Evaluation“ direkt mit den Ergebnissen der hier aufgeführten ähnlichen Publikationen vergleichen zu können. Zum Vergleich werden lediglich Ergebnisse herangezogen, die in Relation zu den hier durchgeführten Untersuchungen stehen.

Bereits im Jahr 2006 wurden durch Moser et al. [69] die Grenzen der statischen Analyse aufgezeigt und dargelegt, dass selbst höher entwickelte Signaturen zur Detektion von Malware nicht zukunftsweisend sind. Hierbei wird Obfuskation durch die Einbringung von opaken Konstanten⁴⁴ durch einigermaßen zuverlässiges Binary Rewriting realisiert. Auch der Kontrollflussgraph wurde bei der Obfuskation berücksichtigt und war ein entscheidendes Merkmal bei der Evaluation der Malware Evasion, da die speziell betrachteten Malware Detektionsmethoden auf dem Kontrollflussgraph basierten und das hierfür eingesetzte Werkzeug IDA Pro diesen aufgrund der Obfuskation nicht generieren konnte. Durch die beschriebene Technik konnte zudem mit drei Malware Samples bei vier ausgewählten kommerziellen Virenscannern Malware Evasion mit einer Quote von 66 % verzeichnet werden. Hinsichtlich der Programmgröße wurde in der Referenzarbeit eine Zunahme um 237 bis 471 % gemessen, welche zu dem hier bestimmten Mittelwert von 386,33 % eine hohe Affinität aufweist. Die Performanz hingegen fällt bei der Technik der Referenzarbeit mit einer Zunahme der Ausführungsdauer von nur 50 bis 100 % deutlich besser aus. Die Detektion durch Virenscanner lässt aufgrund des Alters der Referenzarbeit und der spezifischen Auswahl der Samples und Scanner keinen aussagekräftigen Vergleich zu.

Aktuellere Publikationen zur Malware Evasion behandeln verstärkt Mechanismen aus dem Bereich des maschinellen Lernens und unterscheiden sich primär bei der eingesetzten Lernmethode beziehungsweise beim Lernziel. Als Aktion werden häufig möglichst kleine und wenige, nicht invasive und randomisierte Modifikationen an Malware Samples in Form einer nativ ausführbaren Datei vorgenommen. Zudem wird öfters die Anwendung von Adversarial Attacks untersucht, welche die Malware Evasion durch gezielte Generierung von Störungen (sogenannten Perturbationen) in einem Sample vorsieht. Diese Perturbationen werden von einem auf Grundlage von maschinellem Lernen arbeitender Scanner nicht erwartet, wodurch falsches Verhalten auslösbar ist. Eine Beispielanwendung hierfür wurde durch Castro et al. [107] vorgestellt, mit welcher die Detektionsrate im Schnitt um circa zwanzig Detektionen und sogar vereinzelt auf acht Detektionen reduziert werden konnte. Diese Wirkung wird durch die Thread-basierte Obfuskation nicht erreicht. Es ist jedoch zu beachten, dass bei der Anwendung von Binary Rewriting generell von einer stärkeren Abnahme der Detektionen auszugehen ist. So konnte bereits die Änderung eines einzigen, unbedeutenden Bytes⁴⁵ in der Codesektion im originalen binären Sample des Programms Alina⁴⁶ die Anzahl der Detektionen von 56 auf 44 reduzieren. In [108] wurde hingegen eine Abnahme der Detektionen um 5 bis 12 er-

⁴⁴ Konstantenberechnung zur Laufzeit, deren statische Lösung ein NP-schweres Problem darstellt, siehe auch Abschnitt „Kombination mit anderen Obfuskationsmethoden“

⁴⁵ Interrupt-Byte „CC“

⁴⁶ malwares/Source/Original/Alina/Alina.zip → Spark.exe

reicht, was teilweise auch mit der Thread-basierten Obfuskation möglich war. Weitere Konzepte dieser Art können in [9, 39, 40, 106, 108, 110] gefunden werden.

Während die bereits genannten Publikationen minimal invasives Binary Rewriting zur Obfuskation nutzen, existieren auch etliche Untersuchungen, die den Quellcode für den Transformationsprozess verwenden. Junod et al. [45] entwickelten einen allgemeinen Obfuskator auf Basis von LLVM. Die Art der Integration der Transformationen ist mit dem in dieser Arbeit vorgestellten Verfahren vergleichbar. Choi et al. [9] setzten sogar Obfuskation direkt auf Quellcodeebene um, indem auf Grundlage eines eigens entwickelten Parsers entsprechende Modifikationen durchgeführt wurden. Dabei wurde bei der Generierung von obfuskierter Malware Varianten maschinelles Lernen eingesetzt, während das Ergebnis kontinuierlich anhand einer Ähnlichkeitsbewertung angepasst wurde. Eine Prüfung mit Malware Scannern fand hingegen nicht statt. Die Ergebnisse dieser Referenzarbeit sind mit den Ergebnissen in dieser Arbeit nicht vergleichbar, weil der Fokus der praktischen Untersuchungen stärker auf der Anzahl der benötigten Iterationen für eine ausreichend niedrige Ähnlichkeitsbewertung lag.

Die der hier vorgestellten Obfuskation ähnlichsten Techniken werden von Omar et al. in [50] und der Folgearbeit [84] beschrieben. Als Unterschied zwischen dieser Arbeit und den genannten Publikationen ist hervorzuheben, dass der Fokus von Omar et al. stärker auf der Obfuskation zum Schutz von Software in „feindlicher“ Umgebung liegt und Malware beziehungsweise Malware Evasion in diesem Zusammenhang nicht betrachtet wird. Als feindliche Umgebung werden hierbei Computersysteme bezeichnet, die sich nicht (vollumfänglich) unter der Kontrolle des Eigentümers der Software befinden und potentiell Angriffen ausgesetzt sind (siehe auch Abschnitt „Obfuskation“). Insbesondere wird ein Augenmerk auf Systeme im Kontext von Cloud-Computing gelegt, weshalb bei der Evaluation besonders die Performanz des transformierten Programms auf mehreren Prozessoren unter Berücksichtigung der Hardware Caches als Engpass im Vordergrund steht. Hinsichtlich des Algorithmus wird ebenfalls auf Basis von Basic Blocks gearbeitet, jedoch werden diese gruppiert auf eine beliebige Anzahl von neuen Threads aufgeteilt⁴⁷, wodurch auch Softwarediversität erreicht werden soll.

Im Folgenden werden die Ergebnisse aus [84] mit denen dieser Arbeit verglichen. Hier wurden insgesamt zwei minimalistische Programme für die Evaluation genutzt, die intensive Berechnungen durchführen. Es ist zu beachten, dass in der Referenzarbeit die Ergebnisse in Abhängigkeit von der Anzahl der Threads, auf welche die Basic Blocks aufgeteilt werden, dargestellt sind. Hierbei erfolgte eine Aufteilung auf maximal fünf Threads, sodass der für diese Konfiguration angegebene Höchstwert beim Vergleich herangezogen wird.

- **Längste gemeinsame Teilsequenz:** Diese Metrik wird in Verbindung mit der Formel $\text{Ähnlichkeit} = \text{Länge}(LCS(Seq_1, Seq_2)) \div \text{Länge}(Seq_1)$ (wobei Seq_1 die nicht obfuskierter Variante darstellt) zur Berechnung der Ähnlichkeit verwendet. Auf dieser Basis wird eine maximale Ähnlichkeit von 2 % proklamiert, im Vergleich zu den hier ermittelten Werten in Höhe von 75 %. Ohne die Glaubwürdigkeit dieser Angabe diskreditieren zu wollen, wird aufgrund dieses extremen Unterschieds angenommen, dass vielmehr die längste gemeinsa-

⁴⁷ Sofern die Anzahl an Gruppen mit der Anzahl an Basic Blocks identisch gewählt wird, entspräche dies der Vorgehensweise der in dieser Arbeit vorgestellten Implementierung.

me Teilzeichenfolge für die Bestimmung dieses Werts genutzt wurde⁴⁸. Im Vergleich konnte hier ein Mittelwert von circa 6 % ermittelt werden. In Verbindung mit der minimalistischen Gestaltung der Evaluationsprogramme in der Referenzarbeit und der dadurch geringen Größe von konstanten Daten wäre dieser Wert dann nachvollziehbar.

- **Ausführungszeit:** Aufgrund des Cloud-Computing Kontextes wurde bei den Messungen zur Performanz zwischen der Ausführung auf demselben Prozessor-Die oder verschiedenen Prozessor-Dice und demselben Prozessorkern oder verschiedenen Prozessorkernen unterschieden. Gemäß der hier angewendeten Konfiguration erfolgt der Vergleich mit den Werten für die Ausführung auf demselben Die und demselben Kern. Diesbezüglich wurde für die Programme jeweils ein Anstieg von ungefähr 4.000 % beziehungsweise 9.000 % verzeichnet. Diese Werte sind ähnlich zur den hier getätigten Messungen für das Programm mit der BBP Formel in Höhe von 14.000 %, welches hinsichtlich der Komplexität mit den Programmen der Referenzarbeit am ehesten vergleichbar ist. Bei einer Erhöhung der Komplexität und Anzahl der Threads in der Referenzarbeit wäre ebenfalls von einem steigenden Performanzverlust auszugehen.
- **Anzahl der Instruktionen:** Gemäß der aus dem Diagramm abgelesenen Werte konnte ein Anstieg von etwa 30 auf 600 Instruktionen verzeichnet werden, was einer Zunahme von etwa 2.000 % entspricht. In den hier durchgeführten Messungen konnte lediglich eine maximale Zunahme um ungefähr 2.000 % festgestellt werden. Beim Vergleich dieser Metrik muss jedoch erneut angemerkt werden, dass in der Referenzarbeit keine näheren Angaben zur Bestimmung dieser Werte vorhanden sind und in einem Programm mit leerer *main*-Funktion bereits mehr als 100 Instruktionen vorhanden sind. Zum Vergleich hat das Programm mit der BBP Formel als nicht obfuskierte Variante insgesamt 732 Instruktionen, abzüglich 108 Instruktionen ergibt dies 625, was mit den Werten der Referenzarbeit nicht vergleichbar wäre.

Eine andere Publikation die ein ähnliches Konzept, aber ein anderes Ziel aufweist, ist D-TIME von Pavithran et al. [111]. Anstatt die gestückelte Funktionalität in Threads auszuführen, werden die Instruktionen in einen anderen bereits existierenden und infizierten Prozess eingeschleust und ausgeführt. Anschließend wird das Ergebnis an einen Koordinator zurück übermittelt. Die ausführbaren Dateien werden hierbei in kleinere Stücke unterteilt und mit Verwaltungsinstruktionen angereichert, sodass diese anschließend durch verteilte Emulatoren in den infizierten Prozessen ausgeführt werden können. Der Schwerpunkt liegt hierbei auf der Laufzeitobfuskation. Der Performanzverlust wurde anhand der zusätzlichen Prozessorlast während des Einsatzes der Obfuskation untersucht, weshalb kein Vergleich der Ergebnisse möglich ist.

Als weitere Untersuchungen mit hoher Ähnlichkeit können die Shadow Attacks von Ma et al. [49] und Algorithmen zum Hinzufügen von gutartigen Systemaufrufen wie von Rosenberg et al. [112] genannt werden, insbesondere wenn diese beiden Methoden kombiniert werden würden. Die Technik in [112] benutzt einen Wrapper realisiert durch API-Hooking um gezielt gutartige Systemaufrufe in eine Sequenz bösartiger Systemaufrufe zu injizieren, wohingegen in der hier vorgestellten Obfuskation die Injektion ein Nebenprodukt der eigentlichen Obfuskation ist und anstatt API-Hooking

⁴⁸ Zumal keine näheren Angaben zur Bestimmung des Werts dargelegt werden, wie Anwendung von Compileroptimierung oder Stripping der Binärdateien.

Threads verwendet werden. Der Unterschied der Thread-basierten Obfuskation zu [49] besteht primär in der Tatsache, dass hierbei Prozesse und nicht Threads verwendet werden. Außerdem wird als Ziel vorrangig Malware Evasion bezüglich dynamischer, verhaltensbasierter Detektion mit Sandboxen unter der Annahme untersucht, dass Sandboxen durch die Aufteilung der Funktionalität auf mehrere Prozesse die Sequenz der Systemaufrufe nicht bestimmen können. Hinsichtlich der Performance deutet das in der Referenzarbeit präsentierte Diagramm auch auf eine kostspielige Zunahme der benötigten Ausführungsdauer hin. Dementsprechend wird in dieser Arbeit ebenso wie hier darauf verwiesen, dass die meisten Malware Entwickler dennoch Malware mit besseren Evasion-Eigenschaften bevorzugen würden, sofern die Aufgabe nicht zeitkritisch sei.

Abschließend sind im Kontext dieser Arbeit noch die Aussagen aus [109] zu beachten. In dieser Untersuchung wurde konstatiert, dass zahlreiche Publikationen Mängel bezüglich Experimenten mit Malware aufweisen würden und eine Auseinandersetzung mit diesem Thema als äußerst komplex einzustufen sei. Hierbei wurden vier Felder definiert, die für die Wissenschaftlichkeit solcher Experimente von entscheidender Bedeutung sind. Zunächst wird die Verwendung von korrekten Datensätzen gefordert, was in dieser Arbeit bedauerlicherweise aufgrund der beschränkten Anzahl von zur Verfügung stehenden Quellcodes nicht erfüllt werden kann.

Im nächsten Schritt wird Transparenz bei der Durchführung erwartet, das bedeutet, die verwendeten Systeme, Werkzeuge und Proben sind möglichst detailliert zu beschreiben (Version, Datum, *et cetera*.) und die Gründe für falsche Detektionen oder fehlende Detektionen sind zu analysieren. Sofern relevant für die Nachvollziehbarkeit oder Reproduzierbarkeit, wurden die Versionen der verwendeten Hilfsmittel der Durchführung angegeben, insbesondere beim Korpus, den verwendeten Compilern und den Betriebssystemen. Auch der Zeitpunkt der Durchführung der Untersuchung von Detektionen durch kommerzielle Scanner wurde aufgrund der adaptiven Eigenschaften im Bereich der Malware Detektion angegeben. Auf die konkrete Angabe der Version von weniger veränderlichen Werkzeugen wie Ghidra oder GDB wurde verzichtet. Die Gründe für unerwartete Ergebnisse wurden bestmöglich im Zeitrahmen für die Erstellung der Arbeit untersucht und präsentiert. In einigen Fällen ist dies zudem nicht möglich, wie beim Black-Box-Verfahren bezüglich der Detektion durch kommerzielle Scanner.

Im Weiteren wird an rationales Vorgehen appelliert, also möglichst realistische Untersuchungsumgebungen zu schaffen und keine Ergebnisse ohne ausreichende Grundlage zu verallgemeinern. Dies umfassend umzusetzen erfordert einen sehr hohen Aufwand und würde bei Beachtung und Untersuchung jeder erdenklichen Besonderheit den Umfang vieler Arbeiten überschreiten. In dieser Arbeit wurden die Untersuchungen nach bestem Wissen und Gewissen durchgeführt und im Sinne der zuvor beschriebenen Transparenz zur Nachvollziehbarkeit möglichst genau beschrieben. Dass hierbei trotzdem Fehler unterlaufen sind oder es bessere Alternativen gegeben hätte, ist letztlich nicht auszuschließen.

Zuletzt wird empfohlen, Sicherheitsmaßnahmen bei den Experimenten vorzunehmen und die Implikationen bei der Betrachtung der Ergebnisse miteinzubeziehen. Dies trifft vornehmlich auf die Kompilierung der Samples sowie die Unterabschnitte „Verhaltensbasierte Detektion“ und „Detektion durch kommerzielle Scanner“ zu. Aufgrund des Alters und der Tatsache, dass die Malware beim vorliegenden Quellcode besonders im Hinblick auf Command-and-Control-Server nicht konfiguriert ist, kann die Nutzung von Quellcode gegenüber von binären Samples als sicherer eingestuft

werden. Vor der Kompilierung wurden jegliche ausführbare Dateien aus dem Projekt entfernt. Zur Kompilierung wurde sofern möglich auf die Nutzung von automatisierten Erstellungstools verzichtet. Zudem wurde den Virtuellen Maschinen kein Internetzugriff gewährt und als Hostsystem kam zur Verringerung der Wahrscheinlichkeit von Sandbox Escapes ein Linux OS zum Einsatz⁴⁹. Das Hostsystem und die Virtuellen Maschinen wurden nach der Durchführung der Untersuchungen vernichtet. Abschließend wurden die Kompilate innerhalb eines möglichst engen Zeitraums beim gewählten Online-Dienst VirusTotal eingereicht, um eine eventuelle Adaption der Thread-basierten Obfuskation durch die Scanner mit Einfluss auf folgende Einreichungen auszuschließen.

5.2 Ansätze für Gegenmaßnahmen

Die offensichtlichste Gegenmaßnahme gegen Obfuskation ist die Deobfuskation. Der Einsatz von generischen Deobfuskatoren beziehungsweise Codenormalisierung ist aufgrund der stark invasiven Eigenschaften der implementierten Obfuskation in diesem Kontext nicht zielführend. Vielmehr ist die Erstellung eines gezielten Deobfuskationsalgorithmus notwendig. Die ersten Implikationen hierbei wurden bereits im Unterabschnitt „Widerstandsfähigkeit“ vorgestellt.

Im Folgenden wird von einem realistischen Szenario ausgegangen, sodass der Obfuskationsalgorithmus bei der Implementierung einer Deobfuskationsroutine nicht bekannt ist und somit durch Reverse Engineering Methoden rekonstruiert werden muss. Eine weitere entscheidende Schwierigkeit bei der Deobfuskation ist die Problematik hinsichtlich des Binary Rewritings. Wohingegen die Obfuskation ausgehend des Quellcodes auf Basis der LLVM IR stattfindet, ist als Ausgangspunkt bei der Deobfuskation die ausführbare Datei im Binärformat ohne unterstützende Symbole definiert. Zum aktuellen Zeitpunkt wird davon ausgegangen, dass nach umfangreichen Modifikationen an einer Binärdatei diese nicht mehr ausführbar ist und somit die dynamische Analyse mittels eines Debuggers nach dieser Maßnahme ausscheidet (siehe auch Einleitung des Kapitels „Evaluation“).

Entsprechend dieser Überlegungen ist als Ziel der Deobfuskation vielmehr die Aufbereitung eines Programms zur verbesserten statischen Analyse zu definieren. Insofern die statische Analyse maschinell erfolgen soll, um beispielsweise Signaturen zu generieren oder weiterführende Informationen zu gewinnen, könnte dies relativ einfach auf Basis der disassemblierten Maschineninstruktionen erfolgen. Alternativ ist die Übersetzung in eine andere Sprache wie die LLVM IR, Pseudocode oder auch in C (wie bei Ghidras Decompiler) denkbar, um dem Entwickler das Lesen und Verstehen des Codes zu erleichtern sowie bereits vorhandene Werkzeuge zur integrierten Analyse und Umgestaltung zu nutzen.

Sofern der Entwickler einer Deobfuskationsmethode das allgemeine Prinzip der Thread-basierten Obfuskation verstanden hat, nämlich dass Threads gestartet werden, welche jedoch über Synchronisierung mit Hilfe von Semaphoren seriell abgearbeitet werden, muss zunächst erkannt werden, in welcher Reihenfolge die Threads abgearbeitet werden. Hierfür können ausgehend vom ersten Thread, welcher anhand des fehlenden Wartepunktes am Anfang der Funktion bestimmt werden kann, durch die Zuordnung des benutzten globalen und deshalb eindeutigen Semaphors die Nachfolger bestimmt werden. Angesichts der hierdurch gewonnenen Informationen ließe sich bereits ein verwertbarer Kontrollflussgraph erstellen. Anschließend könnten die verteilten Instruktionen wieder zusammengeführt werden, indem die Funktionen zum *Sperren eines Semaphors* (oder auch Warte-

49 Betrifft insbesondere die Untersuchungen zur verhaltensbasierten Detektion.

punkte) entfernt werden und die Funktionen zum *Freigeben eines Semaphors* durch den entsprechenden Nachfolger ersetzt werden. Ferner werden die Instruktionen zum Neustarten des Threads (sofern sich die Instruktionen in einer Schleife befanden) obsolet und können ebenfalls entfernt werden. Hierbei ist lediglich zu beachten, dass gegebenenfalls vorhandene Funktionen im Zusammenhang mit Parallelität des ursprünglichen Quellcodes bei der Behandlung ausgespart werden, was sich anhand von Mustern leicht erkennen lässt (Position, Argumente, *et cetera*).

Nach diesen Maßnahmen bleiben nur die bei der Obfuskation in den globalen Kontext verlagerten Variablen. Die Umwandlung von globalen Variablen in lokale kann, insbesondere bei der Darstellung in einer Hochsprache, als optional betrachtet werden, sofern die Variablen beziehungsweise Speicherbereiche eindeutig zugeordnet und gegebenenfalls benannt werden können. Hinzu kommt, dass nicht mehr mit Sicherheit rekonstruiert werden kann, ob die Variable ursprünglich lokal oder global war. Ist dennoch eine Deobfuskation erwünscht, kann diese gegebenenfalls mit Hilfe gängiger Compileroptimierungsalgorithmen erfolgen⁵⁰.

Bei Betrachtung der hier skizzierten Vorgehensweise zur Deobfuskation fällt auf, dass diese sich stark an den vier Phasen der Obfuskation in umgekehrter Reihenfolge orientiert. Im ersten Schritt wird die Verlagerung durch Bestimmung der Synchronisationsmechanismen durchgeführt, welche anschließend entfernt werden (Phase drei und vier). Im zweiten Schritt können die in der Datenfluss- und Initialisierungsphase entstandenen Artefakte entfernt werden (Phase eins und zwei).

Neben den statischen Gegenmaßnahmen und dem anfangs ausgeschlossenen Ziel der dynamischen Analyse mit einem Debugger soll an dieser Stelle noch berücksichtigt werden, dass weiterhin die dynamische Analyse über die Aufzeichnung der Systemaufrufe erfolgen kann. Auch wenn die Ergebnisse hierdurch weniger detailreich ausfallen als bei der statischen Deobfuskation, ist die Umsetzung deutlich einfacher. Es müssen lediglich die hinzugekommenen Systemaufrufe im Zusammenhang mit der hinzugefügten Parallelität bestimmt und gefiltert werden. Bei den Systemaufrufen handelt es sich konkret um *NtCreateThread*, *NtDuplicateObject* und *NtClose*.

Abschließend kann (wie im Unterabschnitt „Widerstandsfähigkeit“) festgestellt werden, dass der Aufwand bei der Deobfuskation überschaubar ist. Entscheidend ist jedoch die Erweiterbarkeit dieser Obfuskation sowie die Möglichkeit zur unkomplizierten Kombination mit anderen Obfuskationsmethoden. Da bei diesem „Katz-und-Maus-Spiel“ bereits durch geringfügige Anpassungen Schwierigkeiten bei der Deobfuskation entsprechend der skizzierten Vorgehensweise entstehen können (beispielsweise ist die Erkennung des ersten Threads anhand des fehlenden ersten Wartepunktes ungünstig, sobald ein falscher Wartepunkt eingefügt wird), ist als eigentliche Leistung die Kreation eines möglichst generischen, automatisierten Algorithmus (gegebenenfalls unter Zuhilfenahme von Prinzipien des maschinellen Lernens) anzusehen (siehe auch Abschnitt „Weitere Optimierungen“).

5.3 Kombination mit anderen Obfuskationsmethoden

Eine Kombination der hier vorgestellten Obfuskation mit weiteren Obfuskationsmethoden ist weitestgehend uneingeschränkt möglich. Deshalb sollen im Folgenden nur diejenigen Obfuskationstechniken betrachtet werden, bei denen die höchste Effektivität zu erwarten ist. Des Weiteren kann

⁵⁰ An dieser Stelle sei auf die Funktion *ValueIsOnlyUsedLocallyOrStoredToOneGlobal* des Optimierungsdurchlaufs *GlobalOpt* des LLVM Projekts hingewiesen [117].

die Obfuskation zusätzlich vor oder nach der Thread-basierten Obfuskation erfolgen oder in den beschriebenen Algorithmus integriert werden.

In der verwandten Untersuchung von Omar et al. [50] wurde bereits die offensichtlichste Obfuskationsoptimierung zur Verhinderung der Gewinnung der Ausführungsreihenfolge gemäß Abschnitt „Ansätze für Gegenmaßnahmen“ wie folgt umrissen: „[...] additional protection would hide these accesses, for example by accessing the variables through pointers (the insight is that alias analysis is a difficult problem), or by reusing the same location when we can prove they are never used at the same time. Static reverse engineering is therefore much more difficult, if not impossible.“ Hiermit ist erstens gemeint, dass auf die Variablen für die Synchronisierung (in unserem Fall die Semaphore) über zur Laufzeit berechnete Zeiger zugegriffen wird, anstatt den (auf Ebene der Maschinsprache) reservierten Speicherbereich direkt in der betroffenen Instruktion zu inkludieren. Zweitens wird vorgeschlagen, diese Variablen auch mehrfach zu nutzen, sofern sichergestellt wird, dass die Variable zum Zeitpunkt einer Leseoperation auch den vorgesehenen Wert beinhaltet.

Beide Vorschläge beruhen auf dem Prinzip, dass die Bestimmung von bestimmten Werten bei der statischen Analyse ein unentscheidbares Problem beziehungsweise ein nicht in Polynomialzeit lösbares Problem darstellen kann. Hinsichtlich des zweiten Vorschlags sind etliche Möglichkeiten vorstellbar, bei denen jedoch eventuell recht komplexe Analysen notwendig werden können. Ein einfaches Beispiel ist die mehrfache Nutzung einer globalen Variablen, also die Zuordnung von mehr als einer lokalen Variablen zu einer globalen Variablen, nach entsprechender Prüfung der jeweiligen Schreib- und Lesezugriffe der ursprünglichen Variablen. Komplexere Prüfungen werden zum Beispiel bei mehrfacher Nutzung eines Semaphors notwendig. In diesem Zusammenhang ist auch die Parallelisierung von unabhängigen Funktionalitäten denkbar, wie es teilweise auch im Allgemeinen zur Optimierung von Software eingesetzt wird [113], nur in diesem Kontext mit dem Ziel der Obfuskation.

Die Berechnung der Zeiger zur Laufzeit ist, wie im Zitat angedeutet, ein Aliasing Problem (also die Schwierigkeit, die dadurch entsteht, dass auf Speicherbereiche über verschiedene Wege zugegriffen werden kann) beziehungsweise konkreter ein Problem der Zeiger Analyse [114]. Dieses Problem entsteht hinsichtlich der statischen Analyse vornehmlich, wenn diese Zeiger zu Laufzeit berechnet werden. Wenn diese Berechnung absichtlich und ohne Notwendigkeit durch einen Obfuskator eingebracht wird, so entspricht dies dem Konzept der opaken Variablen⁵¹ (englisch: opaque Variable), welches sich nicht auf Zeiger beschränkt. Ein triviales Beispiel ist die deterministische Berechnung eines Zeigers mit Konstanten, was jedoch bei jeder gängigen Compileroptimierung aufgelöst werden würde und hinsichtlich Obfuskation deshalb als äußerst ineffektiv einzustufen ist.

Um diese Methode effektiv zu gestalten ist die Nutzung von nicht deterministischen Vorgängen und/oder komplexen Kodierungen der Berechnungseingaben üblich. Da das Prinzip zur Generierung einer opaken Variable durch Reverse Engineering rekonstruiert und in einen Deobfuskationsalgorithmus integriert werden kann, wurde durch Moser et al. [69] ein NP-vollständiges Problem bei der Generierung eines zu nutzenden Werts eingebaut, sodass die statische Bestimmung dieser Werte zu einem NP-schwerem Problem wird.

51 Unter den Begriff „opake Variable“ fällt auch der Begriff „opake Konstante“, obwohl sich variabel und konstant im Allgemeinen ausschließen, weil die Berechnung mit einer Variablen erfolgt, das Ergebnis aber konstant bleibt.

Diese generierten, opaken Variablen werden häufig als logisches Prädikat bei der Auswertung einer Sprungbedingung eingesetzt, um den Kontrollfluss zu verschleiern. Im Kontext der Thread-basierenden Obfuskation dürfte der Anwendungsfall vorwiegend bei der Verschleierung der Speicheradressen der neuen Funktionen liegen, um beispielsweise die Erstellung von Funktionsaufrufgraphen oder das Bestimmen der Funktionen bei der Deobfuskation zu erschweren. Auch die Verschleierung der Speicherzugriffe über Zeiger auf die Parallelisierungsmechanismen (Thread-Identifizierer und Semaphore-Identifizierer), um das Feststellen der Ausführungsreihenfolge zu behindern, ist möglich. Diese Anwendung würde sich besonders bei der Widerstandsfähigkeit der Obfuskation durch die Erhöhung des Rechenaufwands bemerkbar machen, sodass diese Kombination als *umfassend* eingestuft werden könnte (siehe auch Abbildung 3).

Laut [2] sei die Fähigkeit zum Einsatz opaker Variablen, deren Bestimmung durch einen Deobfuskator entsprechend schwierig sind, eine große Herausforderung und der Schlüssel zu einer höchst widerstandsfähigen Transformation. Die Wichtigkeit dieses Konstrukts wird in der referenzierten Arbeit häufig hervorgehoben. Die Möglichkeiten sind auch hier unbegrenzt und werden an dieser Stelle deshalb nicht weiter vertieft.

Eine weitere effektive Erweiterung entspringt ebenfalls aus einer grundlegenden Beschreibung von Omar et al. [50] unter Referenzierung von [2], die dort jedoch auch nicht umgesetzt wurde. Auf die hier beschriebene Obfuskation würde es wie folgt angewandt werden: Neben einer neuen Funktion wird eine zusätzliche neue Funktion mit (zufälligen) Instruktionen ohne Seiteneffekte und ohne Nutzen für die eigentliche Aufgabe des Programms erstellt und der ersteren Funktion (für die eigentliche Aufgabe des Programms) durch Synchronisierung mit denselben Semaphoren parallel geschaltet. Insofern in die Funktionen Systemaufrufe eingebaut werden, kann Mimikry (siehe auch Abschnitt „Malware Evasion“) realisiert werden, sodass die Systemaufruf-Sequenzen, die bei der dynamischen Analyse gewonnen werden, kaschiert werden (ähnlich dem API-Hooking von Rosenberg et al. [112]).

Da die hohe Anzahl an Systemaufrufen in Verbindung mit Parallelität und die sich dadurch ergebende monotone Struktur recht auffällig ist und deshalb bei heuristischer Detektion zu einer hohen Erkennungsrate führen kann, kann die primitivste Form der Obfuskation an dieser Stelle hilfreich sein, nämlich das Einfügen von nutzlosem Code. Wird eine umfangreiche Funktion eines anderen Programms genommen, so verändert, dass keine Seiteneffekte zu erwarten sind, und die Instruktionen in diesem Programm so untergebracht, dass diese korrekt ausgeführt werden, gehen die für die Obfuskation entscheidenden Instruktionen unter. Dieses Beispiel erfordert eine komplexe Analyse, weniger komplizierte Vorgehensweisen sind denkbar.

5.4 Weitere Optimierungen

Im Folgenden sollen Optimierungen, Erweiterungen oder auch Alternativen umrissen werden, die nicht notwendigerweise die Metriken der vorgestellten Obfuskation beeinflussen. Im Unterabschnitt „Entscheidungen und Einschränkungen“ wurden bereits die Schwierigkeiten bei rekursiven Funktionen und Funktionen mit variabler Parameteranzahl erläutert, wobei die Implementierung zur Unterstützung dieser Spezialfälle den Funktionsumfang des Programms erweitern würde. Hierbei kann der Effekt einer Optimierung grob in die Kategorien, Obfuskation, Funktionsumfang, und Zuverlässigkeit unterteilt werden.

5.4.1 Obfuskation

Da die Obfuskation als Compileroptimierungsdurchlauf realisiert wurde, liegt es nahe, allgemeine Compileroptimierungen bei der Anwendung miteinzubeziehen. Dies kann unterschiedliche Auswirkungen insbesondere auf die Obfuskationsmetriken haben und bewegt sich in den meisten Fällen im Spannungsfeld zwischen Wirksamkeit, Widerstandsfähigkeit und Kosten. Ebenfalls besteht die Möglichkeit, die Compileroptimierungen vollständig zu deaktivieren, was aufgrund der dadurch entstehenden höheren Informationslast eventuell mit einer Erhöhung der Wirksamkeit verbunden wäre, jedoch die Kosten in fast jeder Hinsicht ungemein in die Höhe treiben würde. Auch bei der Widerstandsfähigkeit gilt im Allgemeinen: Sofern eine geeignete Deobfuskation umgesetzt wird, ist mit der größeren und strukturierteren Menge an Informationen eine akkuratere Rekonstruktion des Untersuchungsgegenstands möglich.

Werden die Optimierungsdurchläufe aktiviert, so ist von einer Reduzierung der Kosten auszugehen. Wie sich die Durchläufe gesamt oder einzeln auf die Obfuskation auswirken ist ohne Untersuchung schwer einzuschätzen. Auch von Einfluss durch die Reihenfolge der Transformationen ist auszugehen. Besonders sollte darauf geachtet werden, dass Optimierungen, die nach der Obfuskation angewandt werden, die Wirksamkeit der Obfuskation nicht dezimieren.

Als Beispiel kann sich vergegenwärtigt werden, dass bei der Obfuskation Funktionen zur Parallelisierung wie *pthread_create* in ein Programm eingebaut werden, die einen Rückgabewert emittieren. Ohne Compileroptimierung wird dieser Rückgabewert als Stackvariable abgelegt, obwohl dieser Wert anschließend nicht weiter verwendet wird. Diese Speicherinstruction ist somit obsolet und vermindert die Performanz eines Programms, auch wird die Wirksamkeit der Obfuskation durch diese im Muster wiederkehrende Instruction nicht erhöht. Bei aktivierten Compileroptimierungen wird diese Instruction entfernt. Auf der anderen Seite konnte (zumindest bei Ghidra) festgestellt werden, dass bei deaktivierten Compileroptimierungen der erste Aufruf von *pthread_create* als letzte Instruction einer erkannten Funktion behandelt wird, wodurch benutzerdefinierte Anpassungen notwendig wären, die eine erhöhte Obfuskationswirksamkeit implizieren.

Des Weiteren sind Optimierungen der Obfuskation in vielen Formen möglich. Beispielsweise kann der Speicher für die Variablen der Funktionen auf dem Stack allokiert werden, anstatt diese in den globalen Raum zu verlagern oder im globalen Raum anzulegen, und die Zeiger als Argument dem Thread übergeben werden. Dies wurde von Omar et al. [84] für die Lösung der Problematik bei rekursiven Funktionen umgesetzt. Hinsichtlich der Erstellung der Semaphore ließe sich sogar eine Schleife realisieren. Diese Methode und der Zugriff auf Ressourcen über als Argument übergebene Zeiger würden die statische Analyse dadurch erschweren, dass die Zuordnung der bei Wartepunkten und Freigabepunkten eingesetzten Semaphore aufgrund des Aliasing Problems nicht mehr ohne Weiteres möglich ist (vergleiche Abschnitt „Ansätze für Gegenmaßnahmen“).

Als weitere möglicherweise entscheidende Performanzoptimierung sei die bereits erwähnte Aussparung von rechenintensiven Funktionen beziehungsweise Funktionalitäten genannt. Dies kann manuell erfolgen, aber auch eine automatische Erkennung von stark frequentierten Instruktionsbereichen kann durch Softwareprofilierung erfolgen, woraufhin die Ergebnisse geeignet auf die Transformation angewandt werden könnten. Zudem wurde bei den Untersuchungen festgestellt, dass vorrangig das häufige Neustarten eines Threads zur Erhöhung der Laufzeit eines Programms beziehungsweise zur Verminderung der Performanz führt. Hinsichtlich dieses Aspekts ließe sich eine signifikante

Performanzoptimierung durch eine Modifikation bei der Behandlung von mehrfach durchlaufenen Basic Blocks gemäß der vorliegenden Implementierung erreichen. Anstelle von sich selbst neu startenden Threads wie im Unterabschnitt „Verlagerung“ beschrieben, erfolgt diese Lebenserhaltungsfunktion durch den Sprung an den Anfang der Funktion. Hierdurch verbleibt als einziger negativer Einfluss bei der Abarbeitung der Funktionalität die Synchronisation durch Semaphore und der damit verbundene Threadwechsel. Bei Ausweitung der Betrachtung von der Funktion auf das gesamte Programm lässt sich erkennen, dass die Effektivität dieser Optimierung bei mehrfach durchlaufenen Basic Blocks mit aufgerufenen obfuskerten Funktionen begrenzt ist. Dieses Hindernis kann jedoch durch aggressives Codeinlining umgangen werden. Durch die dezimierte Anzahl der neu gestarteten Threads ist hingegen von einem negativen Einfluss auf die Wirksamkeit der Obfuskation auszugehen, welcher in Relation mit den Performanzvorteilen abgewogen werden muss.

Eine andere Idee, die etwas stärker vom eigentlichen Entwurf abweicht, entstand durch eine Nachfrage des Betreuers dieser Arbeit. Diese Idee basiert auf dem Entwurfsmuster sogenannter Thread Pools. Anstatt für jede Funktionalität (je nach Einteilungsgröße) einen Thread zu starten, wird eine definierte Anzahl von Threads gestartet, die aus einer Warteschlange (die Parallelität unterstützt) die jeweiligen Funktionszeiger beziehen und aufrufen. Hierbei sind weitere Anpassungen denkbar, wie die Nutzung von Spinlocks oder die Semaphore-Identifizierung zugeordnet zu den jeweiligen Funktionen geeignet zu hinterlegen und die Synchronisation in den wenigen Threads im Pool anstelle in den ausgelagerten Funktionalitäten zu realisieren. In den ursprünglichen Funktionen würden somit keine Threads sondern lediglich diese Warteschlange erstellt werden, die an sich oder deren Erstellung weiterführend obfuskert werden könnte. Hinzu kommt, dass einige Schwierigkeiten wie das Ressourcenproblem beseitigt wären. Bei Untersuchungen wurde durch die exemplarische Implementierung der beschriebenen Alternative indessen festgestellt, dass der Aufwand zur Verwaltung einer solchen Warteschlange höher ist als das repetitive, naive Erstellen eines neuen Threads. Bei weiterführender Performanzoptimierung dieses Konzepts läge schließlich eine Lösung vor, bei welcher nur ein Platzhalter mit der nächsten auszuführenden Funktionalität benötigt wird, ein Thread die Funktionalität ausführt und gemäß Ausgang der Teilfunktionalität der nächste Funktionszeiger in dem Platzhalter abgelegt wird. Entsprechend dem Paradigma, dass Kosten und Wirksamkeit einer Obfuskation in einem Spannungsverhältnis stehen, würden hierbei die Kosten zu Lasten der Wirksamkeit reduziert werden. Im Gegensatz wäre hinsichtlich der Auffälligkeit dieses Design jedoch durch das Wegfallen des Musters bestehend aus den vielen Systemaufrufen besonders wirkungsvoll.

Es existieren viele weitere Möglichkeiten, die einen Einfluss auf die Obfuskation haben können. Als Beispiel kann die Obfuskation einfach mehrfach angewandt werden. Auch die Verwendung von Userspace Threads kann zu relevanten Änderungen führen. Als eine andere ausgefallene Möglichkeit zur Erhöhung der Wirksamkeit bei Programmen, bei denen bereits Parallelisierungsmechanismen vorhanden sind, ist die Eingliederung der bereits vorhandenen Systemaufrufe in die neu hinzugefügten Systemaufrufe, sodass für die Unterscheidung zwischen ursprünglichem und neuem Aufruf komplexere Analysen notwendig werden.

Ebenfalls kann Polymorphie durch Randomisierung des Obfuskationsprozesses erreicht werden. Als Parameter für die Randomisierung kann die Reihenfolge bei der Abarbeitung der Funktionen und/oder Basic Blocks dienen. Dieses Vorgehen ist ansatzweise mit einer gängigen Obfuskation der Datenanordnung zu vergleichen und führt dazu, dass die Position von Funktionalität, die anhand des

Quelltext bestimmt werden könnte, sich überall in der Datei befinden kann, was vorrangig Auswirkungen auf signaturbasierte Detektionsmethoden hat.

Abschließend ist in diesem Kontext nochmals die Wahl der Einteilungsgröße zu erwähnen, also anstatt die Funktion in Basic Blocks aufzuteilen, die Aufteilung grober (mehrere Basic Blocks in eine neue Funktion) oder feiner (bis zu einer Instruktion pro neuer Funktion) vorzunehmen. Neben der möglichen Unterstützung der Polymorphie durch die Randomisierung dieser Größe führt eine gröbere Einteilung zu einem Performanzgewinn, da weniger Threads erstellt werden. Eine kleinere Einteilung führt zu einer Erhöhung der Wirksamkeit, da mehr Informationen hinzugefügt werden und zusammenhängende, detektierbare Funktionalitäten gespalten werden.

5.4.2 Funktionsumfang

Unter Funktionsumfang werden in diesem Kontext die umfangreiche Unterstützung von speziellen (Sprach-)Konstrukten und weitere Fähigkeiten, die die Nutzbarkeit der Obfuskationsmethode verbessern, verstanden. Neben den eingangs erwähnten rekursiven Funktionen und Funktionen mit variabler Parameteranzahl steht noch die Unterstützung von Ausnahmebehandlungen aus, welche in der Programmiersprache C++ Anwendung finden können. Wie bereits festgestellt, ist das Problem bei der Fortführung der Abwicklung der Ausnahmebehandlung in den nächsthöheren Funktionen zu finden. Dies wird in der LLVM IR durch eine sogenannte Resume-Instruktion dargestellt, welche aufgrund der Abkopplung vom ursprünglichen Thread fehlschlägt. Dementsprechend ist der Austausch dieser Instruktion durch eine Funktionalität notwendig, welche der aufrufenden Funktion signalisiert, dass und an welcher Stelle eine Ausnahme eingetreten ist. An dieser Stelle wäre auch das Semaphore der aufrufenden Funktion freizugeben. Nachdem die Bereinigung in der ursprünglichen Funktion durchgeführt wurde, würde anstelle eines gewöhnlichen Returns die abgefangene Ausnahme erneut ausgelöst werden, indem die Ausnahmeparameter aus den entsprechenden globalen Variablen an die *throw*-Funktion übergeben werden.

Eine der nützlichsten Funktionen, die den Anwendungsfall dieser Arbeit bei weitem übersteigt, ist die Fähigkeit, ausführbare Dateien im Binärformat in die LLVM IR zurückzuübersetzen, welche wiederum in eine ausführbare Datei kompiliert werden könnte, nachdem das Programm in der Zwischendarstellung modifiziert wurde. Hierzu wurde bereits im Kapitel „Evaluation“ die Problematik des Binary Rewriting erläutert und ein Einblick in den aktuellen Stand geboten. Die Möglichkeit zur Automatisierung dieses Vorgangs, unter der Bedingung, dass diese weitestgehend fehlerfrei funktioniert, inklusive der Modifikationen wäre als Forschungsgegenstand für wissenschaftliche Untersuchungen eine erhebliche Bereicherung.

Zum Schluss soll, entgegen der für diese Arbeit verallgemeinerten Definition, der Begriff „Metamorphie“, wie er in der Wissenschaft verwendet wird, im Zusammenhang mit der Obfuskation betrachtet werden. Metamorphie meint die reflexive Veränderung der Statik von Maschineninstruktionen eines Programms durch das Programm selbst, meist nach jeder Ausführung beziehungsweise Infektion. Dies kann auf unterschiedliche Weisen erfolgen, zum Beispiel durch minimal invasive Substitution von Instruktionen mit Äquivalenten, durch direkte, invasive Transformationen eines Shellcodes, also von Maschinencode, bei dem keine Binary Rewriting Probleme vorhanden sind [42], oder auch durch die Übersetzung des (überschaubaren) Maschinencodes in eine interne Zwi-

schensprache, anschließender Modifikation und Rückübersetzung (ähnlich dem zuvor beschriebenen Prinzip mit der LLVM IR).

Da die hier vorgestellte Obfuskation aufgrund der invasiven und umfangreichen Auswirkungen nicht nach dem Prinzip dieser Beispiele funktionieren kann, wäre die Anwendbarkeit auf eine andere Weise umzusetzen, welche sich gegebenenfalls mit den beschriebenen Vorgehensweisen kombinieren lässt. Beispielsweise besteht die Möglichkeit für eine Funktion eine definierte Anzahl von neuen Funktionen mit entsprechend groß ausgelegtem Speicherbereich für mehrere Instruktionen anzulegen. Während der metamorphischen Transformation erfolgt die geeignete Verlagerung von (zufällig) ausgewählten Maschineninstruktionen in die neue Funktion mit der entsprechenden Erstellung eines Threads und der Freigabe-/Wartepunkte. Unbenutzte Bytes können hierbei mit nutzlosem Code und/oder unbedingten Sprunginstruktion überbrückt werden. Als weniger komplexe Alternative ist auch die Vertauschung der Speicherbereiche der globalen Variablen denkbar.

5.4.3 Zuverlässigkeit

Wie im Unterabschnitt „Initialisierung“ bezüglich der Funktion zur *Synchronisation mit einem Thread* oder zum *Freigeben der Semaphorressourcen* beschrieben, müssen die im Zusammenhang mit der Parallelität für die Obfuskation allokierten Ressourcen freigegeben werden, um undefiniertes Verhalten bei Fehlgeschlagenen Systemaufrufen zu vermeiden. Diese Maßnahme genügt jedoch nicht, um die Zuverlässigkeit eines beliebig komplexen Programms zu gewährleisten, da bei einem ausreichend langen Aufrufpfad trotzdem das Ressourcenlimit erschöpft sein kann.

Nun bestünde die Möglichkeit, bei jedem Systemaufruf den Rückgabewert zu prüfen und bei einem Fehler eine Behandlungsroutine auszuführen. Aufgrund der durch die Prüfung jedes Systemaufrufs zusätzlich entstehende Last und der Komplexität einer geeigneten Behandlungsroutine bietet sich jedoch eine wegen weniger Seiteneffekte vorzuziehende Alternative an, bei der die maximale Anzahl der zu einem Zeitpunkt aktiven Threads während des Transformationsdurchlaufs (oder auch als Analyseschritt vor der Transformation) bestimmt wird und bei Überschreiten eines definierbaren Schwellwerts entsprechende Maßnahmen ergriffen werden. Die Anzahl kann durch die Summe der Threads aus allen vorangegangenen Funktionsaufrufen im längsten Aufrufpfad zur aktuell betrachteten Funktion ermittelt werden, als einfachste Maßnahme könnte eine Warnung ausgegeben oder automatisch die Einteilungsgröße erhöht werden. Hierbei ist jedoch zu berücksichtigen, dass die Ermittlung des längsten Pfads ein NP-schweres Problem ist [115] und bei hoher Komplexität des Programms die Kosten der Obfuskation (konkret die Transformationsdauer) in die Höhe treiben kann.

Eine andere Möglichkeit, die Wahrscheinlichkeit einer Überschreitung der zur Verfügung stehenden Ressourcen zu verringern, bietet das explizite Setzen der Limitierungen (unter Linux mit *setrlimit* [116]). Dem ist vorausgesetzt, dass die entsprechenden Rechte für diese Operation vorhanden sind. Wird für Threads ein Stackspeicher in Höhe von 8 MiB reserviert, können in einem Programm auf Basis einer 32-bit Architektur ab einer Anzahl von circa 512 Threads keine weiteren Threads erstellt werden. Dieser Wert ergibt sich aus der Beziehung zwischen maximal adressierbarem Speicher von $2^{32} = 4096 \text{ MiB}$ und genanntem Stackspeicher, also $4096 \text{ MiB} \div 8 \text{ MiB} = 512$ ⁵².

52 Hierbei wurden zusätzlich benötigte Speicherbereiche wie Code, Stack, Heap und Zusatzinformationen für die Threads nicht berücksichtigt, da diese bei dieser Größenordnung vernachlässigbar sind.

Der reservierte Speicher in Höhe von 8 MiB wird bei der Größe der Threads im Kontext der Thread-basierten Obfuskation mit hoher Sicherheit nicht ausgeschöpft, sodass sich dieser stark reduzieren lässt. Eine Reduktion auf beispielsweise 1 MiB würde die maximale Anzahl an Threads auf 3600 erhöhen. Bei einem 64-bit System könnten gemäß dieser Rechnung mehr als eine Billion Threads erstellt werden. Diesen Betrachtungen folgt, dass sich durch diese Maßnahme die Limitierungen auf die real vorhandenen Hardwareressourcen reduzieren lassen.

Abschließend gilt an dieser Stelle, dass nicht ausreichend Programme der hier vorgestellten Transformation unterzogen wurden, um die allgemeine Zuverlässigkeit dieser Obfuskation garantieren zu können. Während der Evaluation konnten teilweise Probleme bei Programmen festgestellt werden, die in nicht obfuskiert Form bereits Parallelität einsetzen. Hierbei kann es vorkommen, dass bestimmte Annahmen die bei der Entwicklung existierten, durch die Obfuskation verletzt werden. Solche Konstellationen können beispielsweise entstehen, wenn der Thread-Identifizierer eines aktuellen Threads in einer globalen Variable abgelegt wird, durch die Obfuskation ein neuer Thread gestartet wird, der neue Thread auf ein Ereignis wartet, dieser Wartezustand jedoch über den Thread-Identifizierer durch einen dritten Thread unterbrochen werden soll. Das Unterbrechungssignal würde somit fälschlicherweise an den ersten Thread gesendet werden, was zu undefiniertem Verhalten führt. So ist bei der Nutzung der *signal*-Funktion des Linux OS von Fehlern bei obfuskierten Programmen auszugehen, weil die Effekte der Funktion in einem Multithread-Prozess nicht spezifiziert sind [83].

6 Zusammenfassung und Ausblick

Zu Beginn wurde dem Leser eine kompakte Einführung zu den relevantesten, in dieser Arbeit verwendeten Begriffen geboten und die Beziehungen dieser Begriffe für die weiteren Ausführungen erläutert, um eine entsprechende Grundlage für die folgend vorgestellte Obfuskation und Evaluation herzustellen.

Daraufhin erfolgte die Vorstellung des für diese Arbeit entwickelten Programms unter Beschreibung des Algorithmus. Hierbei wurden auch die Entwurfsentscheidungen erläutert und die ersten Einschränkungen umrissen. Neben der eigentlichen Thread-basierten Obfuskation wird als Voraussetzung eine hiervon separierbare Obfuskation durchgeführt, welche die Verlagerung von Variablen in den globalen Raum vorsieht. Der vorgestellte Algorithmus stellt nur eine Möglichkeit zur Implementierung einer derartigen Obfuskation dar und sollte im folgenden auch als Referenz zu anderen ähnlichen Projekten dienen.

Hiernach schloss sich die Evaluation des entwickelten Programms an. Hierbei ist zu beachten, dass diese aufgrund der recht geringen Anzahl an Samples nicht uneingeschränkt aussagekräftig ist. Nach der Prüfung der semantischen Korrektheit bei der obfuskierten Variante wurden bei den folgenden Untersuchungen formelle Obfuskationsmetriken herangezogen, welche stark an Messungen aus dem Software Engineering Bereich angelehnt, deren Bedeutung bezüglich der beabsichtigen Verschleierung jedoch invertiert sind. Anhand der Ergebnisse musste gefolgert werden, dass aufgrund der hohen Kosten eine gezielte Obfuskation durch präzise Selektion der zu transformierenden Bestandteile eines Programms zu bevorzugen ist. Im Nachhinein erfolgte zudem eine informelle Beschreibung der Auswirkungen der Obfuskation auf das Reverse Engineering anhand einer möglichst realistischen Verfahrensweise. Die Eigenschaften der Obfuskation im Zusammenhang mit der Malware Evasion wurden durch die Betrachtung üblicher Detektionsmechanismen sowie der häufig in wissenschaftlichen Untersuchungen anzutreffenden Prüfung von Samples durch den Online-Dienst VirusTotal untersucht. Es konnte festgestellt werden, dass beim Einsatz der Obfuskation eine Vielzahl an Konsequenzen zu beachtet sind, deren Ursache nicht abschließend ergründbar war. So konnte beispielsweise die Anzahl der Detektionen mit YARA-Regeln durch eine minimale Anpassung erhöht werden, eine weiterführende Diagnose des Ergebnisses wäre ohne umfassende Untersuchungen jedoch nicht möglich. Dennoch lässt sich behaupten, dass signaturbasierte Ansätze für die Erkennung solcher Malware insbesondere bei Kombination mit einer die Datensektionen betreffenden Obfuskation nicht ausreichend ist.

Im letzten Kapitel wurden wissenschaftliche Publikationen vorgestellt, die mit dieser Arbeit verwandt sind, und die festgestellten Ergebnisse soweit möglich verglichen. In den darauf folgenden, eher theoretischen Abschnitten wurden erste Ansätze vorgestellt, mit denen gegen die thematisierte Obfuskation vorgegangen werden könnte. Dabei wurde bereits festgestellt, dass bei der Obfuskation Optimierungspotential vorhanden ist, jedoch minimale Anpassungen die vorgestellten Ansätze schon obsolet werden lassen können. Dementsprechend wurden abschließend Kombinationen mit anderen Obfuskationsformen, wie den opaken Variablen, und weitere Optimierungen vorgestellt, die die Obfuskation an sich und die Zuverlässigkeit erhöhen, aber auch den Funktionsumfang erweitern und somit die Nutzbarkeit verbessern.

Aufgrund der Individualität der vorgestellten Obfuskation wurden die Untersuchungen stärker in der Breite durchgeführt, um zunächst einen ersten Einblick über die Implikationen zu erhalten, die bei der Anwendung einhergehen. Mit diesen Ergebnissen lassen sich aufbauend Untersuchungsfelder priorisieren, welche von einer ausgebildeteren Tiefe gekennzeichnet wären. Für die Zukunft sind dementsprechend weitere Untersuchungen wie der Einfluss der Anwendung von Compileroptimierungen auf obfuskierten Code möglich. Auch etliche Optimierungen oder Erweiterungen der vorliegenden Implementierung sind möglich, von denen einige bereits im Kapitel „Diskussion“ aufgezeigt wurden. Von größtem Nutzen wäre die Entwicklung eines Übersetzers für ausführbare Dateien in eine modifizierbare Zwischensprache mit Möglichkeit zur Rekompilierung. Hinsichtlich der vorgestellten Obfuskation sind die doch recht hohen Kosten in Verbindung mit der Auffälligkeit für einen produktiven Einsatz abzuwägen.

Auch in dieser Arbeit ist schlussendlich festzustellen, dass bei der Masse an im Umlauf befindlicher Software eine hinreichende Analyse von Software nur mit automatisierten Mitteln möglich ist, diese Königsdisziplin aber aufgrund geschickter Obfuskationstechniken und polymorphischen beziehungsweise metamorphischen Generatoren an ihre Grenzen stößt, insbesondere da der Beweis, dass zwei Programme funktionell äquivalent sind, ein unentscheidbares Problem ist [33]. Aktuelle Untersuchungen von neuartigen Methoden zur Malware Detektion im Zusammenhang mit maschinellem Lernen können dieses Ziel unterstützen, aber auch auf Seiten der Gegner wird dieses Instrument für Malware Evasion in Betracht gezogen. Für beide jedoch gilt, dass Techniken basierend auf Künstlicher Intelligenz mit einer fehlenden Greifbarkeit der immanenten Vorgänge verbunden und deshalb nie mit absoluter Sicherheit kontrollierbar sind.

7 Anhang

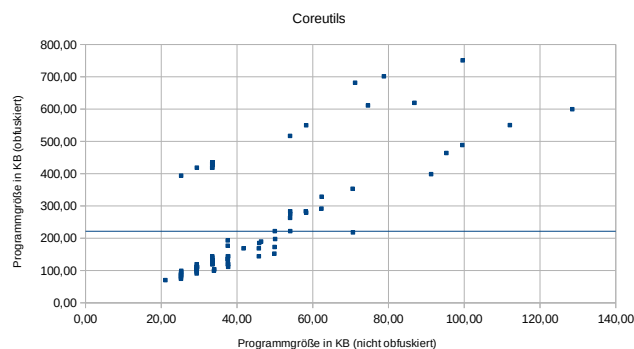
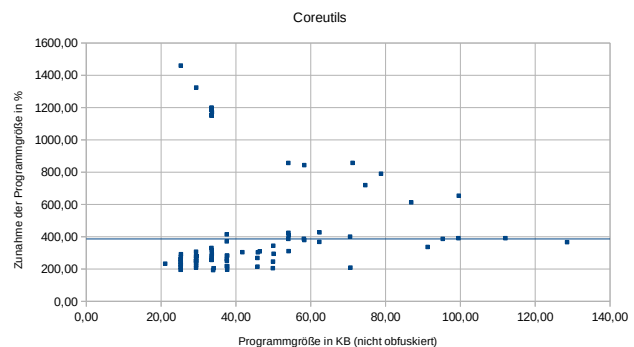
Index

7.1 Messdaten zur Zunahme der Programmgröße.....	78
7.2 Messdaten zur längsten gemeinsamen Teilsequenz.....	79
7.3 Messdaten zur längsten gemeinsamen Teilzeichenfolge.....	80
7.4 Messdaten zur Zunahme der Instruktionen.....	81
7.5 Messdaten zur Transformationsdauer.....	82
7.6 Messdaten zum Speicherplatz.....	83
7.7 Messdaten zur Performanz.....	84
7.8 Messdaten zur Detektion durch kommerzielle Scanner.....	85
Gliederung der beigelegten CD.....	86
Abkürzungsverzeichnis.....	87
Abbildungsverzeichnis.....	88
Codeausschnittsverzeichnis.....	89
Tabellenverzeichnis.....	90
Literaturverzeichnis.....	91

7.1 Messdaten zur Zunahme der Programmgröße

Programmname	Größe in KB (64-bit)		Größe in KB (32-bit)		Zunahme in %	
	n. obfus.	obfus.	n. obfus.	obfus.	64-bit	32-bit
Coreutils						
mkdir	25,25	393,99	20,77	389,46	1460,49	1775,31
seq	29,39	418,62	24,89	414,06	1324,25	1563,71
printf	33,46	434,98	24,88	434,53	1199,83	1646,50
mkdir	33,53	435,04	24,91	430,46	1197,54	1627,94
l	33,45	430,86	28,97	422,23	1188,16	1357,58
who	33,57	426,89	29,02	426,39	1171,71	1369,10
test	33,44	418,57	24,87	414,04	1151,70	1564,68
stdbuf	33,48	418,61	28,98	418,15	1150,32	1342,70
df	71,18	681,62	62,22	713,54	857,66	1046,88
stty	54,00	516,95	49,49	528,77	857,32	968,48
od	54,01	516,96	45,39	532,87	857,19	1073,93
dd	58,26	549,88	53,67	569,82	843,90	961,75
stat	78,78	701,52	61,87	692,73	790,43	1019,61
touch	74,58	611,27	65,92	643,51	719,58	876,20
install	99,57	750,94	82,52	782,99	654,19	848,89
date	86,86	619,45	74,11	651,70	613,12	779,34
ptx	62,34	328,65	53,66	373,18	427,22	595,50
chcon	54,04	283,48	41,31	307,58	424,57	644,54
realpath	37,54	193,30	28,96	205,14	414,96	608,37
chgrp	54,06	275,30	45,42	295,30	409,29	550,22
pr	70,52	353,21	61,86	369,10	400,86	496,62
sort	112,04	550,38	90,80	602,83	391,23	563,88
mv	99,48	488,66	82,47	537,16	391,22	551,35
rm	58,15	283,50	45,42	307,59	387,51	577,28
chown	58,17	283,51	45,42	307,60	387,40	577,18
du	95,29	463,99	86,51	516,62	386,94	497,17
chmod	54,01	262,97	41,30	282,99	386,91	585,28
ln	58,27	279,53	45,48	303,56	379,70	567,51
readlink	37,54	176,91	28,96	188,76	371,31	551,80
tail	62,30	291,74	53,64	328,11	368,32	511,69
vdir	128,52	599,66	107,23	656,13	366,59	511,90
dir	128,52	599,66	107,23	656,13	366,59	511,90
ls	128,52	599,66	107,23	656,13	366,59	511,90
shuf	49,97	222,06	41,32	237,96	344,41	475,90
cp	91,25	398,51	78,34	438,82	336,74	460,12
fmt	33,46	144,11	28,97	164,17	330,75	466,72
cut	33,50	140,06	28,99	151,90	318,13	423,95
numfmt	54,05	222,05	45,42	242,06	310,83	432,90
split	46,36	189,78	41,77	217,93	309,37	421,72
truncate	29,34	119,51	28,96	135,49	307,39	367,85
join	41,72	168,76	33,10	184,69	304,51	457,90
csplit	45,86	185,19	41,32	213,38	303,86	416,42
shred	50,06	197,59	45,47	221,63	294,68	387,40
nl	33,55	131,92	29,03	147,85	293,18	409,26
pwd	25,29	99,11	20,79	106,86	291,93	413,93
uniq	37,58	144,14	28,98	151,90	283,61	424,10
wc	37,66	144,22	29,02	156,03	282,98	437,66
id	33,51	127,78	29,00	131,43	281,31	353,21
tsort	29,33	111,32	20,76	119,10	279,57	473,72
comm	29,34	111,33	24,87	119,11	279,39	378,90
pinky	29,47	111,46	24,93	119,17	278,18	378,05
groups	25,26	94,96	20,78	98,63	275,87	374,74
sum	33,49	123,66	28,99	135,52	269,28	367,49
tr	45,75	168,70	41,26	184,66	268,72	347,50
unexpand	29,37	107,26	24,88	115,02	265,21	362,32
lsc	29,41	107,30	24,90	115,05	264,88	362,11
head	37,55	135,92	33,06	155,98	261,95	371,74
sha1sum	37,57	135,94	33,08	147,80	261,84	346,80
echo	25,19	90,79	20,74	94,50	260,40	355,57
mkfifo	25,22	90,82	20,76	94,52	260,07	355,30
base32	33,46	119,54	28,98	135,50	257,23	367,64
base64	33,46	119,54	28,98	131,41	257,23	353,51
mktmp	33,53	119,62	29,01	127,34	256,76	339,00
chroot	33,54	119,62	24,91	123,25	256,68	394,73
paste	29,33	103,12	20,76	110,90	251,61	434,22
nice	29,34	103,13	20,77	106,82	251,54	414,33
mknd	29,35	103,14	28,97	115,02	251,40	297,05
expand	29,37	103,16	24,87	110,92	251,27	345,96
cat	29,39	103,18	20,81	110,95	251,06	433,22
md5sum	37,57	131,84	28,98	147,80	250,94	409,94
b2sum	49,91	172,86	61,78	213,36	246,32	245,38
uname	25,22	86,72	20,76	90,42	243,91	335,57
nohup	25,25	86,75	20,77	90,43	243,60	335,44
fold	29,35	99,05	24,87	106,82	237,45	329,50
true	21,09	70,30	16,64	74,02	233,38	344,81
false	21,09	70,30	16,64	74,02	233,38	344,81
basename	25,21	82,62	20,75	86,32	227,74	315,96
sleep	25,22	82,62	20,75	82,22	227,66	296,22
cksum	25,22	82,63	20,76	86,33	227,59	315,84
pathchk	25,22	82,63	20,76	90,42	227,59	335,57
env	25,23	82,64	20,76	86,33	227,52	315,84
users	25,26	82,66	20,78	86,34	227,30	315,59
tee	29,37	94,97	20,78	98,64	223,37	374,60
dircolors	37,54	119,54	28,96	127,31	218,39	339,54
uptime	37,69	119,68	29,04	123,28	217,55	324,52
sha256sum	45,76	144,13	37,18	160,09	214,97	330,62
sha224sum	45,76	144,13	37,18	160,09	214,97	330,62
runcon	25,19	78,50	20,74	82,22	211,62	296,34
printenv	25,20	78,51	16,65	78,12	211,56	369,25
yes	25,20	78,51	16,65	82,22	211,56	393,85
dirname	25,21	78,52	16,66	82,22	211,49	393,66
sync	25,23	78,54	20,76	82,23	211,29	296,11
nproc	29,34	90,85	24,86	98,62	209,60	296,65
factor	70,61	218,13	65,87	237,94	208,93	261,21
sha384sum	49,86	152,32	69,94	192,86	205,52	175,73
sha512sum	49,86	152,32	69,94	192,86	205,52	175,73
timeout	34,01	103,70	25,36	107,32	204,94	323,10
expr	37,66	111,45	37,18	156,00	195,96	319,54
unlink	25,20	74,42	16,65	78,13	195,33	369,29
link	25,20	74,42	16,65	78,12	195,30	369,25
hostid	25,20	74,42	16,65	74,02	195,30	344,64
logname	25,21	74,42	16,66	74,03	195,24	344,48
tty	25,22	74,43	16,66	74,03	195,18	344,48
whoami	25,22	74,43	16,66	74,03	195,18	344,48
kill	33,90	99,50	25,29	103,15	193,53	307,83
Mittelwert	43,55	221,54	36,68	235,86	386,33	524,82

Programmname	Größe in KB (64-bit)		Größe in KB (32-bit)		Zunahme in %	
	n. obfus.	obfus.	n. obfus.	obfus.	64-bit	32-bit
Linux						
Exploit.Linux.Interbase	0,23	4,38	0,05	4,17	1789,66	8591,67
bzip2	93,71	638,54	81,09	679,14	581,39	737,53
Net-Worm.Linux.Slapper	37,42	225,90	28,88	225,52	503,63	680,79
bashlite/client	37,61	201,51	28,98	217,42	435,82	650,36
pncan	21,01	107,08	16,59	119,02	409,71	617,48
mirai-botnet/bot	45,61	229,99	41,17	258,29	404,28	527,35
coreutils*	43,55	221,54	36,68	235,86	386,33	524,82
mirai-botnet/loader	21,04	98,92	16,6	110,83	370,15	567,82
bashlite/server	12,76	53,78	8,36	53,45	321,44	539,02
Worm.FreeBSD.Block	8,85	33,49	4,48	33,19	278,48	640,14
KAAL_BHAIRAV	8,53	29,07	8,29	28,80	240,90	247,37
lightaidra	29,22	98,90	24,78	110,82	238,43	347,16
Exploit.Linux.Nhttpd	4,62	12,87	4,38	12,60	178,86	187,76
calc_pi	8,65	16,90	4,27	20,68	95,37	384,54
mettle	804,07	1496,33	839,35	1740,47	86,09	107,36
Virus.Unix.Adrastea	12,55	20,81	12,35	20,58	65,77	66,58
Mittelwert	74,34	218,13	72,27	241,93	399,14	963,61



7.2 Messdaten zur längsten gemeinsamen Teilsequenz

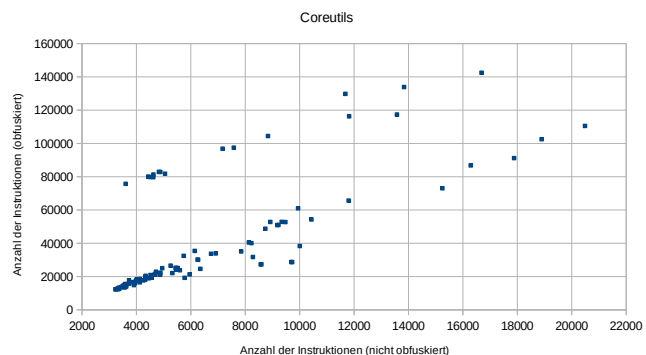
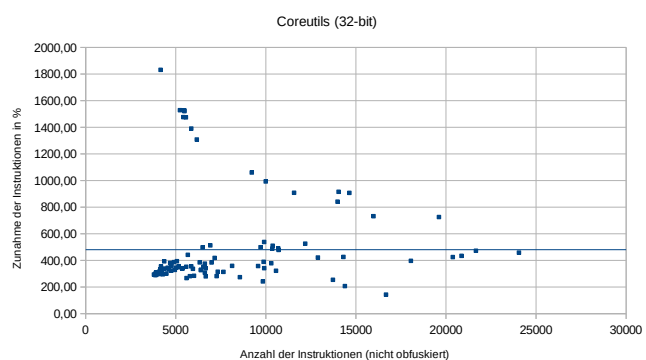
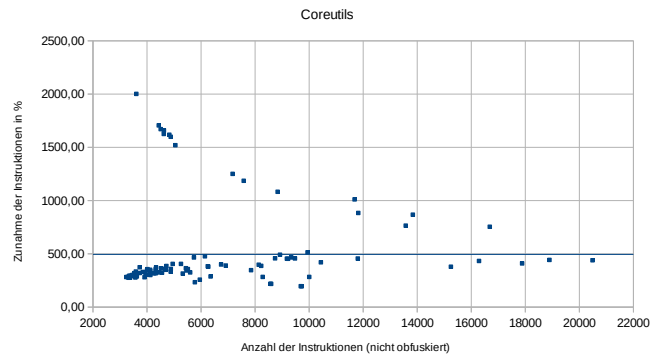
Programmname	Größe in B (64-bit)		Größe in B (32-bit)		LCSSeq in B		Ähnlichkeit		Ähnlichkeit 2	
	n. obfus	obfus.	n. obfus	obfus.	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit
Coreutils										
hostid	31216	80432	22180	79556	25172	17145	50.24	40.82	80.64	77.30
logname	31224	80440	22188	79564	25117	17126	50.12	40.76	80.44	77.19
tty	31232	80448	22188	79564	25061	17092	50.00	40.68	80.24	77.03
whoami	31232	80448	22188	79564	25054	17115	49.98	40.73	80.22	77.14
link	31216	80432	22180	83652	24999	17077	49.89	39.65	80.08	76.99
unlink	31216	80440	22180	83660	24624	17109	49.14	39.72	78.88	77.14
printenv	31216	84528	22180	83652	25114	17188	48.89	39.90	80.45	77.49
expr	43672	117464	42716	161532	34876	31738	48.69	38.21	79.86	74.30
runcon	31208	84520	26276	87748	24954	21246	48.59	44.25	79.96	80.86
sync	31248	84560	26292	87764	24926	20963	48.49	43.64	79.77	79.73
dimame	31224	84536	22188	87756	24797	17037	48.27	38.61	79.42	76.78
kill	39912	105512	30824	108680	31103	23892	47.93	41.28	77.93	77.51
yes	31216	84528	22180	87748	24532	17000	47.76	38.53	78.59	76.65
timeout	40024	109720	30896	112848	31519	24116	47.56	40.84	78.75	78.06
uptime	43704	125696	34572	128812	35048	27575	47.29	41.32	80.19	79.76
env	31248	88656	26292	91860	24764	20997	47.05	42.73	79.25	79.86
basename	31224	88632	26284	91852	24739	21034	47.03	42.81	79.23	80.03
nproc	35360	96864	30396	104156	27484	22446	46.96	39.89	77.73	73.85
factor	76624	224144	71404	243468	61467	58578	46.90	44.43	80.22	82.04
dircolors	43560	125552	34496	132840	34673	27151	46.89	40.11	79.60	78.71
users	31272	88680	26308	91876	24612	20978	46.74	42.67	78.70	79.74
cksum	31240	88648	26292	91860	24565	20969	46.68	42.67	78.63	79.75
true	27104	76320	22172	79548	21207	17256	46.63	41.09	78.24	77.83
false	27104	76320	22172	79548	21204	17254	46.62	41.08	78.23	77.82
sleep	31232	88640	26284	87756	24431	20686	46.43	43.07	78.22	78.70
pathchk	31240	88648	26292	95956	24311	20674	46.20	41.16	77.82	78.63
tee	35384	100984	26316	104172	27495	20011	46.00	38.22	77.70	76.04
uname	31232	92736	26292	95956	24487	20672	45.50	41.16	78.40	78.62
nohup	31264	92768	26300	95964	24262	20487	45.05	40.78	77.60	77.90
expand	35384	109176	30404	116452	27612	23447	44.43	39.40	78.04	77.12
nice	35352	109144	26300	112348	27569	19525	44.38	35.92	77.98	74.24
fold	35368	105064	30404	112356	26999	22013	44.29	37.66	76.34	72.40
echo	31208	96808	26276	100036	24309	20477	44.23	39.94	77.89	77.93
paste	35344	109136	26292	116436	27351	19663	44.04	35.54	77.39	74.79
cat	35408	109200	26340	116484	27326	19569	43.95	35.33	77.17	74.29
mkfifo	31240	96840	26292	100052	24141	20265	43.89	39.51	77.28	77.08
unexpand	35384	113272	30412	120556	27580	23311	43.56	38.50	77.94	76.65
mknd	35368	109160	34500	120548	26789	25674	43.11	39.81	75.74	74.42
chroot	39552	125632	30444	128780	30259	22611	42.93	36.11	76.50	74.27
groups	31280	100976	26308	104164	24107	20319	42.89	38.81	77.07	77.24
base64	39480	125560	34508	136940	30085	25203	42.73	36.66	76.20	73.04
mktemp	39544	125632	34540	132876	30044	26471	42.63	39.07	75.98	76.64
base32	39480	125560	34508	141036	29992	25218	42.60	36.15	75.97	73.08
tac	35424	113320	30428	120580	26972	23229	42.57	38.35	76.14	76.34
pwd	31304	105128	26324	112388	24310	20481	42.38	37.65	77.66	77.80
tsort	35344	117336	26292	124636	27201	19638	42.24	34.31	76.96	74.69
md5sum	43584	137856	34516	153332	32727	25295	42.22	34.73	75.08	73.20
pinky	35488	117472	30460	124700	27131	23377	42.02	37.93	76.45	76.75
comm	35360	117344	30404	124644	26774	23205	41.56	37.69	75.72	76.32
head	43568	141936	38596	161508	32606	27397	41.46	34.70	74.84	70.98
id	39528	133800	34532	136964	30031	25978	41.29	37.77	75.97	75.23
nl	39568	137936	34564	153390	30327	24286	41.05	33.35	76.65	70.26
sha224sum	51776	150144	42708	165620	36080	29206	40.92	34.73	69.68	68.39
sha256sum	51776	150144	42708	165620	36005	29228	40.84	34.75	69.54	68.44
sha1sum	43584	141952	38612	153332	31799	29417	40.43	38.23	72.96	76.19
wc	43672	150232	34552	161560	32658	24672	40.32	33.02	74.78	71.41
sum	39504	129680	34520	141048	28770	23780	40.20	34.08	72.83	68.89
truncate	35352	125528	34492	141020	26775	24821	40.19	35.59	75.74	71.96
tr	51768	174712	46796	190188	38181	33402	40.15	35.41	73.75	71.38
uniq	43592	150160	34516	157436	32300	24973	39.92	33.88	74.10	72.35
sha384sum	55872	158336	75476	198388	37372	52320	39.73	42.76	66.89	69.32
sha512sum	55872	158336	75476	198388	37344	52314	39.70	42.75	66.84	69.31
b2sum	55928	178872	67308	218892	39683	49028	39.68	40.39	70.95	72.84
cut	39512	146072	34524	157436	29756	26266	39.17	35.63	75.31	76.08
csplit	51872	191208	46852	218916	38534	31447	38.69	31.05	74.29	67.12
join	47736	174776	38636	190220	35081	27647	38.41	32.25	73.49	71.56
fmt	39472	150128	34500	169700	29042	23968	37.73	31.32	73.58	69.47
shred	56080	203608	51004	227164	40049	35206	37.48	32.71	71.41	69.03
split	52376	195800	47304	223464	37756	32286	37.28	31.40	72.09	68.25
numfmt	60064	228064	50956	247596	42835	35269	36.60	31.40	71.32	69.21
readlink	43552	182928	34492	194292	32368	25692	36.26	31.38	74.32	74.49
shuf	55984	228080	46852	243492	39574	31523	35.02	29.51	70.69	67.28
cp	97264	404528	83876	444356	69271	59723	34.92	30.94	71.22	71.20
realpath	43552	199312	34492	210676	31919	25023	34.26	29.35	73.29	72.55
ln	64288	285544	51008	309088	45784	35728	33.79	28.45	71.22	70.04
rm	64168	289512	50948	313124	45619	36227	33.47	28.68	71.09	71.11
chmod	60024	268984	46828	288524	42362	32122	33.34	27.63	70.58	68.60
pr	76536	359224	67396	374628	55188	45807	33.28	28.83	72.11	67.97
chown	64184	289528	50956	313132	45263	35289	33.20	27.94	70.52	69.25
tail	68312	297760	59172	333644	47325	39495	33.18	28.11	69.28	66.75
dir	134536	605672	112760	661664	92984	75987	32.57	27.82	69.11	67.39
ls	134536	605672	112760	661664	92984	75987	32.57	27.82	69.11	67.39
vdif	134536	605672	112760	661664	92984	75987	32.57	27.82	69.11	67.39
chgrp	60072	281320	50948	300836	42006	35693	32.31	28.83	69.93	70.06
chcon	60056	289496	46844	313116	42605	32573	32.31	26.90	70.94	69.54
du	101304	470008	92044	522156	69576	61028	31.89	27.84	68.68	66.30
mv	105496	494680	88000	542688	72542	60141	31.75	27.52	68.76	68.34
ptx	68352	334664	59188	378708	47449	39182	31.37	26.17	69.42	66.20
sort	118056	556392	96336	608360	76967	60456	30.03	24.97	65.20	62.76
date	92880	625464	79644	657228	70355	57573	29.19	25.16	75.75	72.29
install	105584	756952	88048	788520	76404	62953	27.03	23.89	72.36	71.50
stty	60016	522968	55020	534300	46964	41611	26.51	24.27	78.25	75.63
touch	80600	617288	71452	649044	58903	50183	26.41	23.30	73.08	70.23
od	60024	522976	50924	538404	45536	35876	25.70	21.67	75.86	70.45
dd	64272	555896	59200	575352	47779	42427	25.28	22.99	74.34	71.67
stdbuf	39496	424624	34516	423684	32457	26471	25.06	21.89	82.18	76.69
test	39456	424584	30404	419572	31938	23639	24.68	20.93	80.95	77.75
stat	84800	707536	67404	698260	60239	49493	24.59	22.81	71.04	73.43
who	39584	432904	34556	431924	31797	27470	24.29	22.49	80.33	79.49
mkdir	39544	441056	30444	435996	31709	23602	24.01	20.49	80.19	77.53
printf	39480	440992	30412	440060	31677	23391	24.01	20.22	80.24	76.91
[39464	436880	34500	427764	31134	27402	23.71	22.56	78.89	79.43
df	77192	687640	67748	719076	54440	45957	23.63	20.82	70.53	67.84
rmid	31264	400008	26300	394996	26202	21826	23.43	21.41	83.81	82.99
seq	35408	424632	30420	419596	28221	23914	23.02	21.17	79.70	78.61
Mittelwert	49562.74	227559.16	42207.28	241387.						

7.3 Messdaten zur längsten gemeinsamen Teilzeichenfolge

Programmname	Größe in B (64-bit)		Größe in B (32-bit)		LCSStr in B		Ähnlichkeit		Ähnlichkeit 2	
	n. obfus	obfus.	n. obfus	obfus.	64-bit	32-bit	64-bit	32-bit	64-bit	32-bit
Coreutils										
factor	76624	224144	71404	243468	12437	12393	9.49	9.40	16.23	17.36
hostid	31216	80432	22180	79556	3630	1115	7.24	2.65	11.63	5.03
logname	31224	80440	22188	79564	3294	1116	6.57	2.66	10.55	5.03
numfmt	60064	228064	50956	247596	7408	7421	6.33	6.61	12.33	14.56
printenv	31216	84528	22180	83652	3158	1115	6.15	2.59	10.12	5.03
dircolors	43560	125552	34496	132840	4237	4205	5.73	6.21	9.73	12.19
tr	51768	174712	46796	190188	5365	5349	5.64	5.67	10.36	11.43
stty	60016	522968	55020	534300	9968	9968	5.63	5.81	16.61	18.12
whoami	31232	80448	22188	79564	2622	1115	5.23	2.65	8.40	5.03
runcon	31208	84520	26276	87748	2606	3926	5.07	8.18	8.35	14.94
link	31216	80432	22180	83652	2471	1115	4.93	2.59	7.92	5.03
tty	31232	80448	22188	79564	2462	1115	4.91	2.65	7.88	5.03
split	52376	195800	47304	223464	4341	4357	4.29	4.24	8.29	9.21
tsort	35344	117336	26292	124636	2742	1115	4.26	1.95	7.76	4.24
fold	35368	105064	30404	112356	2590	1116	4.25	1.91	7.32	3.67
sync	31248	84560	26292	87764	2054	3026	4.00	6.30	6.57	11.51
cksum	31240	88648	26292	91860	2102	2478	3.99	5.04	6.73	9.42
basename	31224	88632	26284	91852	2086	3626	3.97	7.38	6.68	13.80
shred	56080	203608	51004	227164	4214	4507	3.94	4.19	7.51	8.84
cut	39512	146072	34524	157436	2934	2922	3.86	3.96	7.43	8.46
md5sum	43584	137856	34516	153332	2965	2929	3.83	4.03	6.80	8.49
nl	39568	137936	34564	153380	2789	2777	3.78	3.81	7.05	8.03
sha1sum	43584	141952	38612	153332	2965	2945	3.77	3.83	6.80	7.63
unexpand	35384	113272	30412	120556	2358	1690	3.72	2.79	6.66	5.56
expr	43672	117464	42716	161532	2661	2645	3.72	3.18	6.09	6.19
timeout	40024	109720	30896	112848	2452	2468	3.70	4.18	6.13	7.99
env	31248	88656	26292	91860	1943	3306	3.69	6.73	6.22	12.57
tail	68312	297760	59172	333644	5053	5025	3.54	3.58	7.40	8.49
dirname	31224	84536	22188	87756	1814	1115	3.53	2.53	5.81	5.03
dir	134536	605672	112760	661664	9991	9999	3.50	3.66	7.43	8.87
ls	134536	605672	112760	661664	9991	9999	3.50	3.66	7.43	8.87
vdirc	134536	605672	112760	661664	9991	9999	3.50	3.66	7.43	8.87
join	47736	174776	38636	190220	3189	3173	3.49	3.70	6.68	8.21
uniq	43592	150160	34516	157436	2821	2801	3.49	3.80	6.47	8.12
sha512sum	55872	158336	75476	198388	3270	3105	3.48	2.54	5.85	4.11
sha384sum	55872	158336	75476	198388	3269	3105	3.48	2.54	5.85	4.11
sha224sum	51776	150144	42708	165620	3013	2977	3.42	3.54	5.82	6.97
sha256sum	51776	150144	42708	165620	3013	2977	3.42	3.54	5.82	6.97
mktemp	39544	125632	34540	132876	2357	2341	3.34	3.46	5.96	6.78
mknod	35368	109160	34500	120548	2077	3182	3.34	4.93	5.87	9.22
expand	35384	109176	30404	116452	2054	1542	3.30	2.59	5.80	5.07
truncate	35352	125528	34492	141020	2197	2177	3.30	3.12	6.21	6.31
pr	76536	359224	67396	374628	5461	5453	3.29	3.43	7.14	8.09
yes	31216	84528	22180	87748	1686	1115	3.28	2.53	5.40	5.03
chroot	39552	125632	30444	128780	2302	1359	3.27	2.17	5.82	4.46
cp	97264	404528	83876	444356	6365	6357	3.21	3.29	6.54	7.58
head	43568	141936	38596	161508	2455	2350	3.12	2.98	5.63	6.09
comm	35360	117344	30404	124644	2005	2586	3.11	4.20	5.67	8.51
b2sum	55928	178872	67308	218892	3093	3153	3.09	2.60	5.53	4.68
l	39464	436880	34500	427764	4037	4022	3.07	3.31	10.23	11.66
users	31272	88680	26308	91876	1614	2294	3.06	4.67	5.16	8.72
ln	64288	285544	51008	309088	4133	4113	3.05	3.28	6.43	8.06
uname	31232	92736	26292	95956	1629	2862	3.03	5.70	5.22	10.89
dd	64272	555896	59200	575352	5683	5734	3.01	3.11	8.84	9.69
sleep	31232	88640	26284	87756	1558	1116	2.96	2.32	4.99	4.25
csplit	51872	191208	46852	218916	2933	2913	2.95	2.88	5.65	6.22
od	60024	522976	50924	538404	5182	5177	2.92	3.13	8.63	10.17
echo	31208	96808	26276	100036	1589	1573	2.89	3.07	5.09	5.99
id	39528	133800	34532	136964	2102	1949	2.89	2.83	5.32	5.64
pinky	35488	117472	30460	124700	1829	2654	2.83	4.31	5.15	8.71
sort	118056	556392	96336	608360	7249	7237	2.83	2.99	6.14	7.51
cat	35408	109200	26340	116484	1742	1681	2.80	3.03	4.92	6.38
shuf	55984	228080	46852	243492	3150	1717	2.79	1.61	5.63	3.66
test	39456	424584	30404	419572	3600	3588	2.78	3.18	9.12	11.80
wc	43672	150232	34552	161560	2198	1987	2.71	2.66	5.03	5.75
tee	35384	100984	26316	104172	1619	1607	2.71	3.07	4.58	6.11
kill	39912	105512	30824	108680	1749	1866	2.70	3.22	4.38	6.05
nohup	31264	92768	26300	95964	1429	2130	2.65	4.24	4.57	8.10
sum	39504	129680	34520	141048	1894	1115	2.65	1.60	4.79	3.23
nice	35352	109144	26300	112348	1630	1283	2.62	2.36	4.61	4.88
du	101304	470008	92044	522156	5525	5509	2.53	2.51	5.45	5.99
chcon	60056	289496	46844	313116	3285	3261	2.49	2.69	5.47	6.96
fmt	39472	150128	34500	169700	1829	1821	2.38	2.38	4.63	5.28
pathchk	31240	88648	26292	95956	1236	3166	2.35	6.30	3.96	12.04
date	92880	625464	79644	657228	5589	5577	2.32	2.44	6.02	7.00
chown	64184	289528	50956	313132	3117	3101	2.29	2.45	4.86	6.09
mkfifo	31240	96840	26292	100052	1236	1290	2.25	2.52	3.96	4.91
uptime	43704	125696	34572	128812	1662	1510	2.24	2.26	3.80	4.37
gtx	68352	334664	59188	378708	3381	3353	2.24	2.24	4.95	5.66
pwd	31304	105128	26324	112388	1267	2078	2.21	3.82	4.05	7.89
stat	84800	707536	67404	698260	5352	5351	2.18	2.47	6.31	7.94
tac	35424	113320	30428	120580	1363	1351	2.15	2.23	3.85	4.44
base64	39480	125560	34508	136940	1486	1451	2.11	2.11	3.76	4.20
paste	35344	109136	26292	116436	1301	1289	2.09	2.33	3.68	4.90
readlink	43552	182928	34492	194292	1867	2454	2.09	3.00	4.29	7.11
base32	39480	125560	34508	141036	1459	3126	2.07	4.48	3.70	9.06
realpath	43552	199312	34492	210676	1926	1719	2.07	2.02	4.42	4.98
false	27104	76320	22172	79548	896	1230	1.97	2.93	3.31	5.55
true	27104	76320	22172	79548	896	1230	1.97	2.93	3.31	5.55
rm	64168	289512	50948	313124	2653	2641	1.95	2.09	4.13	5.18
chmod	60024	268984	46828	288524	2421	2405	1.91	2.07	4.03	5.14
chgrp	60072	281320	50948	300836	2461	2445	1.89	1.97	4.10	4.80
printf	39480	440992	30412	440060	2437	2417	1.85	2.09	6.17	7.95
seq	35408	424632	30420	419596	2227	2441	1.82	2.16	6.29	8.02
unlink	31216	80440	22180	83660	896	1115	1.79	2.59	2.87	5.03
who	39584	432904	34556	431924	2277	2257	1.74	1.85	5.75	6.53
df	77192	687640	67748	719076	3934	3943	1.71	1.79	5.10	5.82
touch	80600	617288	71452	649044	3789	3789	1.70	1.76	4.70	5.30
install	105584	756952	88048	788520	4741	4719	1.68	1.79	4.49	5.36
nproc	35360	96864	30396	104156	958	1230	1.64	2.19	2.71	4.05
stdbuf	39496	424624	34516	423684	2083	2068	1.61	1.71	5.27	5.99
groups	31280	100976	26308	104164	896	2038	1.59	3.89	2.86	7.75
mv	105496	494680	88000	542688	3007	3007	1.32	1.38	2.85	3.42
mkdir	39544	441056	30444	435996	1693	1497	1.28	1.30	4.28	4.92
rmidr	31264	400008	26300	394996	1156	3101	1.03	3.04	3.70	11.79
Mittelwert	49562.74	227559.16	42207.28	241387.47	3134.97	3117.60	3.18	3.31	6.18	7.29

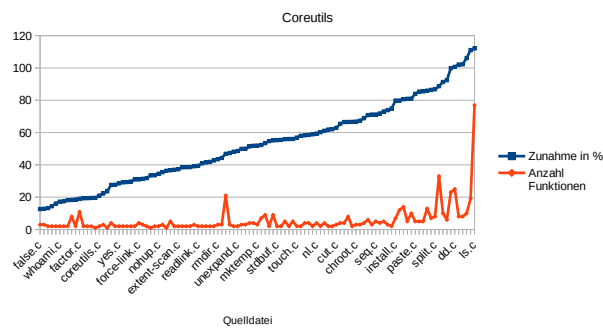
7.4 Messdaten zur Zunahme der Instruktionen

Programmname	Instruktionen (64-bit)		Instruktionen (32-bit)		Zunahme in %	
	n. obfus.	obfus.	n. obfus.	obfus.	64-bit	32-bit
Coreutils						
mkdir	3604	75754	4156	80281	2001,94	1831,69
stdbuf	4434	80090	6167	86838	1706,27	1308,11
seq	4506	79807	5231	85184	1671,13	1528,45
who	4625	81349	5561	87568	1658,90	1474,68
test	4620	79726	5420	85453	1625,67	1476,62
printf	4826	82930	5476	89106	1618,40	1527,21
mkdir	4877	82871	5482	88875	1599,22	1521,21
l	5050	81814	5861	87355	1520,08	1390,45
stty	7167	96840	9218	106986	1251,19	1060,62
od	7578	97477	9993	109259	1186,32	993,36
dd	8834	104475	11567	116596	1082,65	908,01
df	11682	129794	14639	147451	1011,06	907,25
touch	11819	116287	13981	131600	883,90	841,28
stat	13837	133902	14039	142575	867,71	915,56
date	13575	117378	15977	132964	764,66	732,22
install	16684	142497	19610	161949	754,09	725,85
ptx	9940	61031	12186	76252	513,99	525,73
chcon	8921	52807	9904	63244	491,94	538,57
realpath	6142	35428	6917	42398	476,82	512,95
rm	9339	52875	10376	63324	466,17	510,29
readlink	5736	32472	6489	38789	466,11	497,77
chmod	8738	48742	9706	58144	457,82	499,05
chown	9474	52752	10689	63042	456,81	489,78
pr	11800	65642	14309	75287	456,29	426,15
chgrp	9175	50929	10362	61060	455,08	489,27
ln	9219	51042	10704	61941	453,66	478,67
sort	18893	102558	21662	124014	442,84	472,50
vdirc	20486	110535	24051	134366	439,56	458,67
dir	20486	110535	24051	134366	439,56	458,67
ls	20486	110535	24051	134366	439,56	458,67
du	16286	86804	20379	107033	433,00	425,21
tail	10433	54313	12887	67233	420,59	421,71
mv	17881	91205	20862	111497	410,07	434,45
cut	4949	25031	5669	30717	405,78	441,84
fmt	5256	26579	6985	33832	405,69	384,35
csplit	6742	33709	9908	43754	399,99	341,60
shuf	8133	40485	10300	49291	397,79	378,55
split	6914	33897	9569	43808	390,27	357,81
numfmt	8226	40091	9875	48326	387,37	389,38
nl	4718	22926	6659	29356	385,93	340,85
join	6248	30200	7158	37135	383,35	418,79
cat	6268	30078	8127	37310	379,87	359,09
cp	15246	73051	18052	89703	379,15	396,91
pwd	3729	17675	4360	21501	373,99	393,14
dircolors	4336	20469	5062	25014	372,07	394,15
wc	5437	25283	6613	31462	365,02	375,76
truncate	4518	20946	6614	27006	363,61	308,32
base32	4678	21563	6402	27538	360,94	330,15
id	4881	22366	5861	26701	358,23	355,57
tsort	4322	19767	4903	23880	357,36	387,05
uniq	5511	25197	6329	30712	357,21	385,26
paste	4018	18331	4689	22575	356,22	381,45
base64	4688	21272	6391	27292	353,75	327,04
chroot	4695	21147	5582	25246	350,42	352,28
unexpand	4116	18535	5158	23145	350,32	348,72
expand	4002	17886	5007	22306	346,93	345,50
shred	7850	35075	10564	44598	346,82	322,17
nice	4099	18290	4759	21920	345,96	360,60
head	5462	24246	7648	31672	343,90	314,12
pinky	4323	19147	5158	23565	342,91	356,86
cat	4121	18115	4744	22242	339,58	368,84
echo	3586	15561	4179	19010	333,94	354,89
mktemp	4875	21123	5951	26001	333,29	336,92
groups	3848	16536	4600	19867	329,73	331,89
kill	3933	16843	4551	20258	328,25	345,13
md5sum	5597	23820	6539	29593	325,59	352,56
tac	4452	18939	5361	23500	325,40	338,35
comm	4555	19284	5380	23754	323,36	341,52
mkfifo	3728	15666	4349	19082	320,23	338,77
uname	3531	14792	4138	17925	318,92	333,18
mknod	4320	18034	6012	23113	317,45	284,45
timeout	4249	17111	4949	21227	316,83	328,91
fold	4114	17140	5784	22073	316,63	281,62
sum	5323	22097	7266	27762	315,12	282,08
nproc	3951	16030	5597	20565	305,72	267,43
pathchk	3622	14599	4221	17812	303,06	321,99
env	3460	13921	4073	16898	302,34	314,88
tee	4124	16544	4753	20160	301,16	324,15
users	3588	14321	4147	17374	299,14	318,95
basename	3510	13984	4112	16992	298,40	313,23
dirname	3426	13574	4022	16470	296,21	309,50
printenv	3351	13248	3913	16015	295,34	309,28
runcon	3352	13220	3929	16039	294,39	308,22
unlink	3354	13093	3893	15828	290,37	306,58
cksum	3597	14019	4279	16931	289,74	295,68
sleep	3623	14068	4146	16968	288,30	309,26
sha1sum	6352	24635	7329	30417	287,83	315,02
tty	3288	12647	3857	15309	284,64	296,91
yes	3553	13631	4058	16294	283,65	301,53
b2sum	8280	31758	14389	44205	283,55	207,21
factor	10003	38347	13725	48654	283,35	254,49
true	3232	12348	3789	14929	282,05	294,01
false	3235	12348	3788	14929	281,70	294,11
nohup	3907	14867	4483	18008	280,52	301,70
sync	3569	13500	4107	16332	278,26	297,66
whoami	3347	12619	3886	15234	277,02	292,02
link	3374	12704	3920	15344	276,53	291,43
logname	3333	12514	3866	15114	275,46	290,95
hostid	3317	12452	3845	15032	275,40	290,95
uptime	5953	21286	6664	25348	257,57	280,37
expr	5770	19249	8559	32021	233,60	274,12
sha224sum	8567	27345	9835	33653	219,19	242,18
sha256sum	8594	27340	9837	33657	218,13	242,15
sha384sum	9689	28722	16671	40546	196,44	143,21
sha512sum	9723	28720	16676	40538	195,38	143,09
Mittelwert	6712	40759	8224	48236	494,88	480,57



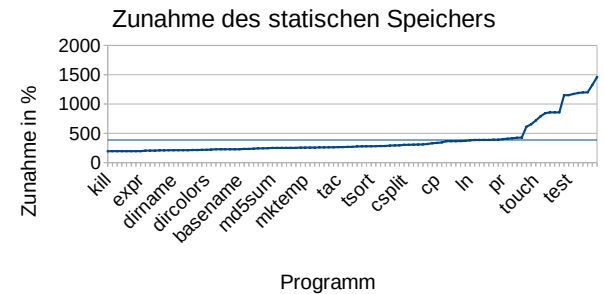
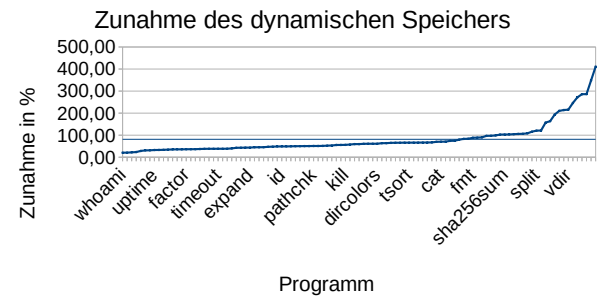
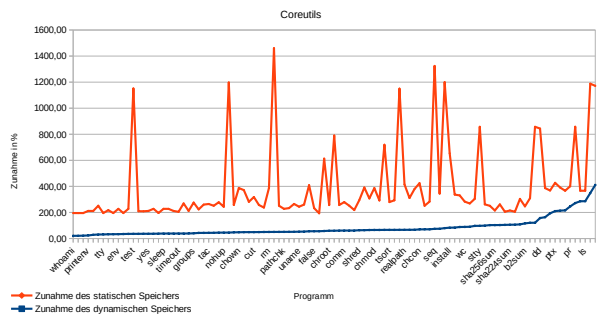
7.5 Messdaten zur Transformationsdauer

C-Quellcode	LOC	Anzahl Funktionen	Kompilierzeit in ms		Zunahme in %
			o. obfus.	obfus.	
false.c	3	3	460,14	518,23	12,62
true.c	81	3	451,46	509,33	12,82
hostid.c	89	2	462,17	523,32	13,23
logname.c	85	2	469,30	536,84	14,39
unlink.c	91	2	474,68	550,85	16,05
whoami.c	93	2	476,71	558,85	17,23
nproc.c	130	2	482,37	566,58	17,46
link.c	96	2	483,87	571,62	18,14
cp-hash.c	165	8	449,65	531,92	18,28
hostdbuf.c	147	2	434,61	515,17	18,54
factor.c	2662	11	3329,74	3960,91	18,96
tty.c	134	2	480,84	573,75	19,32
hostname.c	119	2	480,22	573,27	19,38
dimame.c	137	2	510,14	609,76	19,53
find-mount-point.c	114	1	459,30	549,20	19,57
coreutils.c	208	2	497,07	601,23	20,96
expr.c	1108	3	1949,33	2383,65	22,28
operand2sig.c	94	1	446,12	551,49	23,62
users.c	153	4	574,37	732,78	27,58
sleep.c	150	2	518,93	662,50	27,67
yes.c	129	2	555,51	713,94	28,52
mkfifo.c	183	2	545,13	704,09	29,16
runcon.c	265	2	563,31	728,57	29,34
groups.c	146	2	539,93	700,02	29,65
nice.c	222	2	565,71	740,92	30,97
force-link.c	183	4	476,91	624,70	30,99
basename.c	191	3	562,12	738,33	31,35
sync.c	238	2	562,86	741,86	31,80
make-prime-list.c	231	1	453,18	604,76	33,45
printenv.c	155	2	528,71	705,96	33,53
nohup.c	228	2	553,52	744,46	34,50
cksum.c	314	3	613,72	831,26	35,45
relpath.c	134	1	506,49	690,68	36,37
mkdir.c	297	5	599,55	819,46	36,68
env.c	186	2	582,44	797,86	36,99
extent-scan.c	228	2	553,52	759,98	37,30
chown.c	332	2	593,06	821,43	38,51
chgrp.c	320	2	599,68	830,95	38,56
tee.c	279	2	713,41	988,98	38,63
uptime.c	259	3	649,72	904,22	39,17
readlink.c	179	2	528,76	736,96	39,38
group-list.c	123	2	505,91	713,18	40,97
mknod.c	276	2	634,85	896,38	41,51
expand.c	239	2	634,45	899,96	41,85
pathchk.c	423	2	742,05	1059,26	42,75
mkdir.c	252	3	596,20	855,50	43,48
rm.c	358	3	608,71	878,29	44,29
copy.c	3020	21	2633,62	3867,26	46,84
mv.c	494	3	741,93	1093,02	47,32
kill.c	315	2	709,00	1050,39	48,15
unexpand.c	327	2	710,60	1055,86	48,59
fold.c	310	3	770,67	1155,22	49,90
date.c	604	3	784,18	1176,25	50,00
sum.c	274	4	666,79	1009,50	51,40
pwd.c	395	4	799,19	1213,47	51,84
mktemp.c	351	3	671,31	1020,15	51,96
timeout.c	549	7	683,26	1040,72	52,32
test.c	886	9	1313,63	2016,13	53,48
shuf.c	611	2	1113,52	1722,86	54,72
lbracket.c	3	9	1329,71	2063,38	55,17
stdbuf.c	394	2	759,22	1179,32	55,33
truncate.c	400	2	766,70	1192,01	55,47
lsort.c	569	5	985,61	1505,55	55,92
chmod.c	571	2	892,37	1392,16	56,01
head.c	1095	5	2020,77	3152,91	56,03
touch.c	439	2	729,14	1143,95	56,89
id.c	441	2	798,34	1261,22	57,98
comm.c	500	4	945,94	1497,50	58,31
tac.c	714	4	1019,02	1615,70	58,55
getlimits.c	173	2	1011,30	1607,14	58,92
nl.c	596	4	896,11	1427,81	59,33
uname.c	377	2	621,17	995,38	60,24
dircolors.c	510	4	994,67	1602,45	61,10
realpath.c	279	2	675,98	1094,19	61,87
chcon.c	588	2	866,58	1404,59	62,08
cut.c	610	3	1189,05	1937,12	62,91
who.c	836	4	1260,00	2084,34	65,42
pinky.c	603	4	1013,67	1687,97	66,52
expand-common.c	401	8	679,93	1132,53	66,57
cat.c	768	2	1009,45	1681,75	66,60
chroot.c	431	3	770,91	1286,03	66,82
set-fields.c	321	3	760,79	1273,08	67,34
remove.c	587	4	886,53	1497,21	68,88
chown-core.c	556	6	948,41	1620,10	70,82
du.c	1140	3	1573,76	2693,42	71,15
seq.c	707	5	1138,27	1948,26	71,16
uniq.c	676	4	1026,92	1764,23	71,80
shred.c	1313	5	1545,14	2673,40	73,02
ln.c	619	3	842,70	1465,28	73,88
echo.c	273	2	661,29	1156,23	74,85
install.c	1060	7	1123,49	2020,15	79,81
tr.c	1909	12	2453,17	4411,51	79,83
tail.c	2507	14	3638,28	6574,06	80,69
numfmt.c	1652	5	2146,78	3883,43	80,90
gtx.c	2149	10	4116,59	7453,32	81,06
paste.c	531	5	958,96	1764,16	83,97
printf.c	716	5	1190,38	2206,24	85,34
wc.c	859	5	1233,56	2288,21	85,50
csplit.c	1527	13	1882,12	3518,28	86,94
silly.c	2336	7	2284,97	4259,59	86,42
split.c	1668	8	2328,61	4350,67	86,84
sort.c	4780	33	7241,94	13667,44	88,73
join.c	1200	10	1832,13	3505,13	91,31
cp.c	1223	6	1171,58	2254,31	92,42
od.c	1982	23	2468,16	4933,08	99,87
dd.c	2486	25	2870,10	5758,98	100,65
fmt.c	1030	8	1569,81	3172,92	102,12
df.c	1794	8	2610,83	5281,18	102,28
stat.c	1664	10	4839,02	9973,76	106,11
pr.c	2848	19	3908,20	8248,60	111,06
ls.c	5301	77	5898,76	12509,69	112,07
Mittelwert	712,41	5,45	1166,08	1957,72	52,70



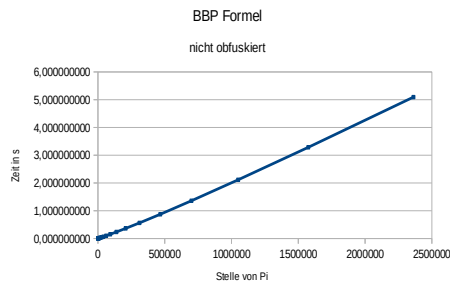
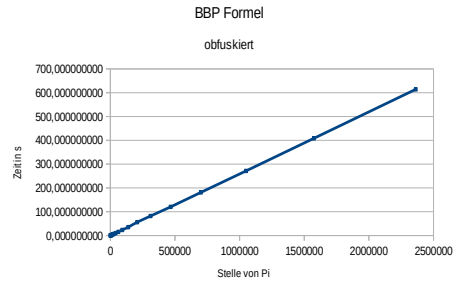
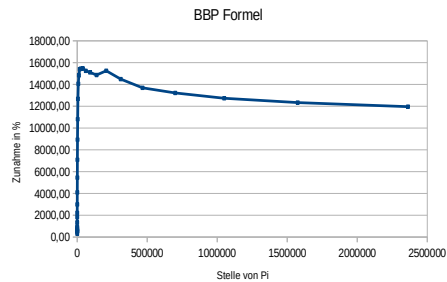
7.6 Messdaten zum Speicherplatz

Programmname	Speichergröße in KB (64-bit)		Speichergröße in KB (32-bit)		Zunahme in %		Zunahme der Programmgröße in %	
	n. obfus.	obfus.	n. obfus.	obfus.	64-bit	32-bit	64-bit	32-bit
whoami	2714,22	3272,00	2393,78	2742,67	20,55	14,57	195,18	344,48
hostname	2673,78	3229,33	2398,89	2870,22	20,78	19,75	195,24	344,48
hostid	2549,78	3109,33	2245,33	2701,78	21,95	20,33	195,30	344,64
grepnterv	2369,33	2922,22	1964,00	2528,00	23,34	28,72	211,56	369,25
sync	2220,44	2951,11	1949,33	2459,58	28,40	33,04	211,29	296,11
psce	2293,58	2952,22	1940,00	2539,58	30,87	30,00	251,54	414,33
lty	2192,44	2876,44	1853,33	2421,33	31,20	30,65	195,18	344,48
uptime	2433,33	3229,33	2139,11	2757,33	32,71	28,90	217,55	324,52
unlrmk	2399,58	3110,22	1973,33	2701,33	32,94	35,95	195,33	369,29
snv	2251,33	3019,11	1956,00	2631,58	33,51	34,54	227,52	315,84
link	2347,56	3156,89	1982,22	2728,00	34,48	37,62	195,30	369,25
cksum	2177,33	2946,67	1829,33	2404,89	35,33	31,46	227,59	315,84
test	2146,22	2912,44	1821,78	2391,11	35,70	42,23	215,70	1564,68
rmcc	2271,58	3084,00	1970,22	2620,00	35,77	32,98	209,60	296,65
factor	2312,44	3150,67	1870,22	2547,11	36,25	36,19	208,93	261,21
yes	2081,33	2838,22	1773,33	2291,11	36,37	29,20	211,58	393,85
basename	2363,78	3214,67	2012,89	2741,33	36,57	36,19	227,74	315,96
expr	2453,78	3368,00	1959,11	2800,44	37,26	42,94	195,96	319,54
sleep	2340,00	3231,11	1965,78	2739,58	38,08	39,36	227,66	296,22
users	2157,33	2981,78	1864,44	2596,22	38,22	38,71	227,30	315,59
dirname	2286,67	3161,33	2004,44	2749,78	38,25	37,18	211,49	393,66
timeout	2480,89	3453,78	2138,67	2932,44	38,41	37,12	204,94	323,10
sum	2140,44	2963,11	1839,11	2375,58	38,43	29,17	269,28	367,49
runcon	2307,58	3199,11	1926,67	2694,22	38,64	39,84	211,62	296,34
groups	2480,89	3175,58	1959,11	2749,78	39,98	39,98	275,87	374,74
tee	2141,33	3057,33	1855,11	2468,44	42,78	33,06	223,37	374,60
mkfifo	2292,89	3282,22	2002,22	2802,22	43,15	39,96	260,07	355,30
tac	2321,78	3332,89	1984,22	2749,78	43,55	37,89	264,88	362,11
expand	2184,00	3111,11	1826,67	2544,44	43,77	44,77	251,27	345,96
pinky	2814,67	4084,89	2468,89	3388,00	45,13	37,23	278,18	378,05
nohup	2340,44	3400,89	1944,89	2802,67	45,31	44,10	243,60	335,44
mkdir	2339,58	3405,78	1985,78	2992,00	45,57	50,67	1197,54	1627,94
base32	2210,67	3254,22	1888,11	2571,78	47,21	47,93	275,87	387,64
chown	2394,22	3542,22	1980,89	2962,22	47,95	49,67	387,40	577,18
readlink	2328,44	3464,00	2006,67	2879,58	48,77	43,50	371,31	551,80
id	2809,33	4185,78	2405,78	3409,78	49,00	41,73	281,31	353,21
cut	2325,78	3467,11	1981,33	2880,00	49,07	45,36	318,13	423,56
base64	2171,11	3245,33	1853,78	2570,11	49,48	38,51	257,23	353,51
fold	2123,11	3183,58	1835,11	2543,11	49,95	38,58	237,45	329,50
rm	2430,22	3552,44	1991,58	3085,33	50,29	54,92	387,51	577,28
md5	2319,11	3490,67	1960,44	2816,00	50,52	51,74	1775,31	362,36
mkmod	2339,11	3524,00	1986,44	2920,00	50,66	46,38	251,40	297,05
pathchk	2323,11	3505,78	2022,67	2884,89	50,91	42,63	227,59	335,57
true	1195,11	1804,44	916,44	1474,22	50,99	60,86	233,38	344,81
unexpand	2169,89	3275,00	1817,78	2619,11	51,32	44,98	265,21	362,36
uname	2146,67	3275,11	1852,67	2703,58	52,57	45,14	243,91	355,57
echo	2121,33	3248,00	1775,11	2657,33	53,11	49,70	260,40	355,57
chgrp	2315,58	3602,67	2006,22	2937,33	55,59	46,41	409,29	550,22
lsc	1195,00	1956,00	888,44	1492,00	55,70	67,93	233,38	344,81
kill	2337,78	3647,11	1984,00	2965,00	56,01	48,99	193,53	307,83
date	2368,00	3731,58	2040,00	3132,44	57,58	53,55	613,12	779,34
chroot	2333,33	3722,22	1980,44	2985,33	59,52	50,74	256,68	394,73
stat	2368,67	3827,11	2043,11	3284,44	59,58	60,96	790,43	1019,61
mktemp	2217,78	3564,89	1981,33	2934,67	60,74	48,12	256,76	339,00
comm	2318,67	3736,00	2021,78	3068,89	61,13	51,79	279,39	378,90
paste	2157,33	3476,44	1828,44	2765,78	61,15	51,26	251,61	434,22
brooks	2286,44	3678,22	1938,89	3045,78	61,29	51,38	218,38	339,54
shred	2304,44	3770,22	1984,00	3065,78	63,61	55,53	294,68	387,40
mv	2388,00	3913,33	2028,89	3428,00	63,87	68,96	391,22	551,35
truncate	2304,89	3806,22	2043,11	3163,11	65,14	54,82	307,39	367,85
chmod	2401,33	3915,11	2016,44	3234,67	65,54	60,41	386,91	585,25
pwd	2118,67	3509,78	1841,78	2972,89	65,66	55,98	291,93	413,93
touch	2406,44	4000,89	2021,78	3264,89	66,12	61,49	719,58	876,20
lsort	2129,33	3537,33	1802,22	2673,78	66,12	48,36	279,57	473,72
nl	2171,33	3520,00	1891,58	2835,58	66,25	49,91	293,18	409,26
stoduf	2367,11	3936,44	1970,67	3199,58	66,30	62,36	1150,32	1342,70
realpath	2304,00	3838,22	1999,11	3144,89	66,59	57,31	414,96	608,37
numfmt	2221,78	3706,67	1838,67	2895,78	66,83	52,60	310,83	432,90
ln	2331,58	3898,67	2000,00	3205,78	67,21	60,29	379,70	587,51
lchcon	2359,11	4005,78	2007,11	3215,58	69,80	60,21	424,57	644,54
cat	2087,11	3557,78	1760,44	2763,58	70,46	56,98	251,06	433,22
uniq	2168,89	3702,22	1814,22	2765,78	70,70	52,45	283,61	424,10
seq	2282,67	3976,44	1937,33	3222,67	74,20	66,35	1324,25	1553,71
shuf	2202,67	3849,78	1870,67	2940,44	74,78	57,19	344,41	475,90
grepref	2322,67	4171,58	1918,22	3253,78	79,60	69,62	1199,83	1646,50
install	2409,78	4412,00	2035,11	3951,11	83,09	74,49	654,19	848,89
cp	2425,78	4458,22	2035,58	3990,22	83,79	76,38	338,74	460,12
fmt	2152,00	4050,22	1828,00	3038,67	88,21	66,23	330,75	456,72
wc	2208,00	4172,44	1833,78	3041,33	88,97	65,85	282,98	437,66
tr	2318,67	4406,67	2016,00	3376,00	90,05	67,48	268,72	347,50
join	2489,78	4745,33	1977,33	3537,78	96,92	78,92	304,51	457,90
stty	2280,22	4517,78	1943,58	3442,22	97,26	77,11	857,32	968,48
head	2207,56	4391,11	1863,11	3167,58	98,91	70,01	261,95	371,74
md5sum	2191,58	4442,67	1839,58	3296,44	102,72	79,20	250,94	409,94
sha256sum	2185,33	4435,58	1799,58	3297,78	102,97	83,36	214,97	330,63
sha1sum	2179,11	4430,67	1867,58	3300,00	103,32	76,70	261,84	346,80
sha512sum	2190,67	4480,00	1881,33	3296,44	104,50	75,22	205,52	175,73
sha224sum	2186,67	4456,00	1858,67	3304,89	105,66	77,81	214,97	330,62
sha384sum	2194,99	4505,78	1872,00	3341,78	106,22	78,51	205,52	175,73
csplit	2406,44	5015,11	2008,44	3776,00	108,23	88,01	303,86	416,42
bdsum	2179,11	4705,33	1898,67	3468,00	115,93	82,65	246,32	245,38
split	2133,78	4705,78	1777,78	3370,67	120,54	89,60	309,37	421,72
dd	2364,44	5226,67	2014,22	3980,89	121,05	97,64	857,19	1073,93
du	2316,89	5954,67	1934,67	4022,22	157,01	107,90	843,90	961,75
du	2505,33	6603,11	2089,33	4759,11	163,56	127,78	386,94	497,17
tail	2220,67	6542,67	1905,33	4366,67	193,31	129,18	368,32	511,69
join	2293,78	7114,67	2014,67	4894,89	210,17	132,54	427,22	595,50
sort	2419,11	7594,22	2087,11	5106,22	213,93	144,66	391,23	563,88
vdif	2909,33	9188,44	2619,11	6421,33	215,83	145,17	366,59	511,90
pr	2303,11	7969,78	2079,11	4825,78	246,04	132,11	400,89	486,62
df	2354,67	9423,11	2227,58	6712,00	271,77	201,32	857,66	1046,88
dir	2315,11	8931,11	1965,78	6041,78	285,77	207,35	366,59	511,90
ls	2300,44	8885,78	1924,44	6025,78	286,26	213,12	366,59	511,90
l	2340,00	10491,11	2028,89	5846,67	348,34	188,17	1189,16	1357,58
who	2374,22	12129,33	2119,11	7013,78	410,89	230,96	1171,71	1369,10
Mittelwert	2291,04	4168,31	1953,80	3216,66	81,27	64,24	386,33	524,82



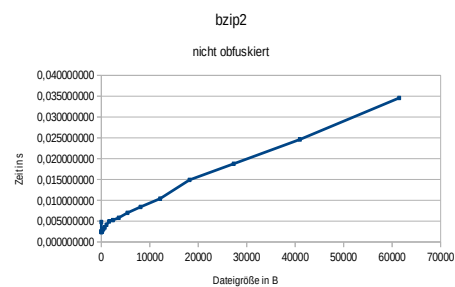
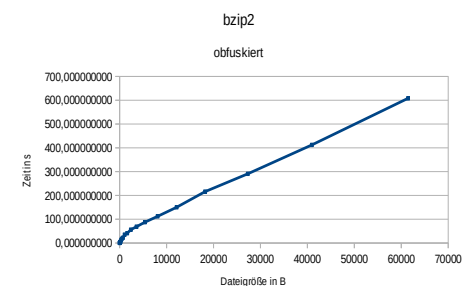
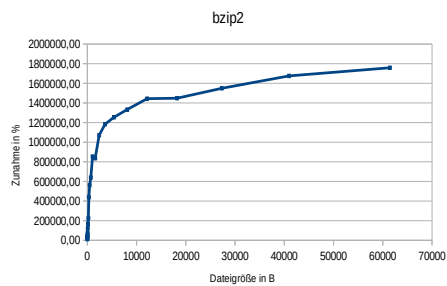
7.7 Messdaten zur Performanz

Stelle von Pi	Zeit in s (n.obfus.)	Zeit in s (obfus.)	Zunahme in %
BBP Formel			
3	0.005960254	0.024601286	312,76
4	0.004060231	0.024433832	501,78
6	0.004091514	0.024830528	506,88
9	0.004124024	0.025531881	519,10
13	0.004097252	0.026640933	550,21
19	0.004102145	0.028144681	586,10
28	0.004100132	0.030653546	647,62
42	0.004122916	0.033793066	719,64
63	0.004097614	0.038587058	841,70
94	0.004336523	0.046121246	963,55
141	0.004183562	0.060228195	1339,59
211	0.004257981	0.082693474	1842,26
316	0.004342512	0.100813662	2221,55
474	0.004504702	0.139289938	2992,01
711	0.004760529	0.199591375	4092,63
1066	0.005091002	0.282691379	5452,77
1599	0.005735300	0.412455751	7091,53
2398	0.006755351	0.610681249	8939,96
3597	0.008285795	0.904186957	10812,49
5395	0.010478442	1.339378231	12682,23
8092	0.014165539	2.006316301	14063,36
12138	0.019970629	2.981917620	14831,52
18207	0.028880856	4.477045320	15401,78
27310	0.043023308	6.691950629	15454,24
40965	0.064694706	10.069013245	15463,89
61447	0.098790332	15.161453343	15247,10
92170	0.152794808	23.231762895	15104,55
138255	0.234906917	35.148376403	14862,68
207382	0.364461206	55.959379521	15254,00
311073	0.561633159	81.930949418	14487,98
466609	0.872195268	120.277280240	13690,18
699913	1.359611054	181.100187033	13220,00
1049869	2.113815314	271.271327185	12733,26
1574803	3.284107594	408.433471968	12336,67
2362204	5.093732853	614.289687188	11959,32
Mittelwert	0,41	52,50	7935,05



Vergleich	LOC	Anzahl Funktionen	n.obfus.	Programmgröße in B	Max Zunahme in %
BBP Formel	149	5	14664	15463.89	15463.89
bzip2	6419	109	99728	175844.95	175844.95
Verhältnis	43.08	21.80	6.80	113,71	

Dateigröße	Zeit in s (n.obfus.)	Zeit in s (obfus.)	Zunahme in %
bzip2			
3	0.004790633	0.543979964	11255,07
4	0.002375573	0.578478769	24251,13
6	0.002377382	0.605409318	25365,38
9	0.002492981	0.665272287	26585,81
13	0.002473247	0.778808151	31389,30
19	0.002443287	0.915160999	37356,14
28	0.002378607	1.132632045	47517,45
42	0.002542177	1.528465914	60024,29
63	0.002514505	1.992998354	79160,07
94	0.002484991	3.074288031	123614,25
141	0.002545452	4.315021060	169418,85
211	0.002742311	6.216829973	226600,40
316	0.002994204	13.150994212	439115,04
474	0.003212326	18.068192220	562364,46
711	0.003483968	22.269940329	639111,97
1066	0.004135571	35.325853981	854095,32
1599	0.004921463	41.161332181	836283,74
2398	0.005248910	56.212206277	1070831,07
3597	0.005809076	68.73032719	1183054,30
5395	0.006992046	87.674209557	1253813,51
8092	0.008490006	112.074534742	1332691,71
12138	0.010403420	150.220639808	1443854,39
18207	0.014906847	215.891147688	1448168,35
27310	0.018770538	290.842936785	1549365,11
40965	0.024608507	412.323353722	1675431,77
61447	0.034578409	608.076865239	1758444,95
Mittelwert	0,01	82,86	650351,69

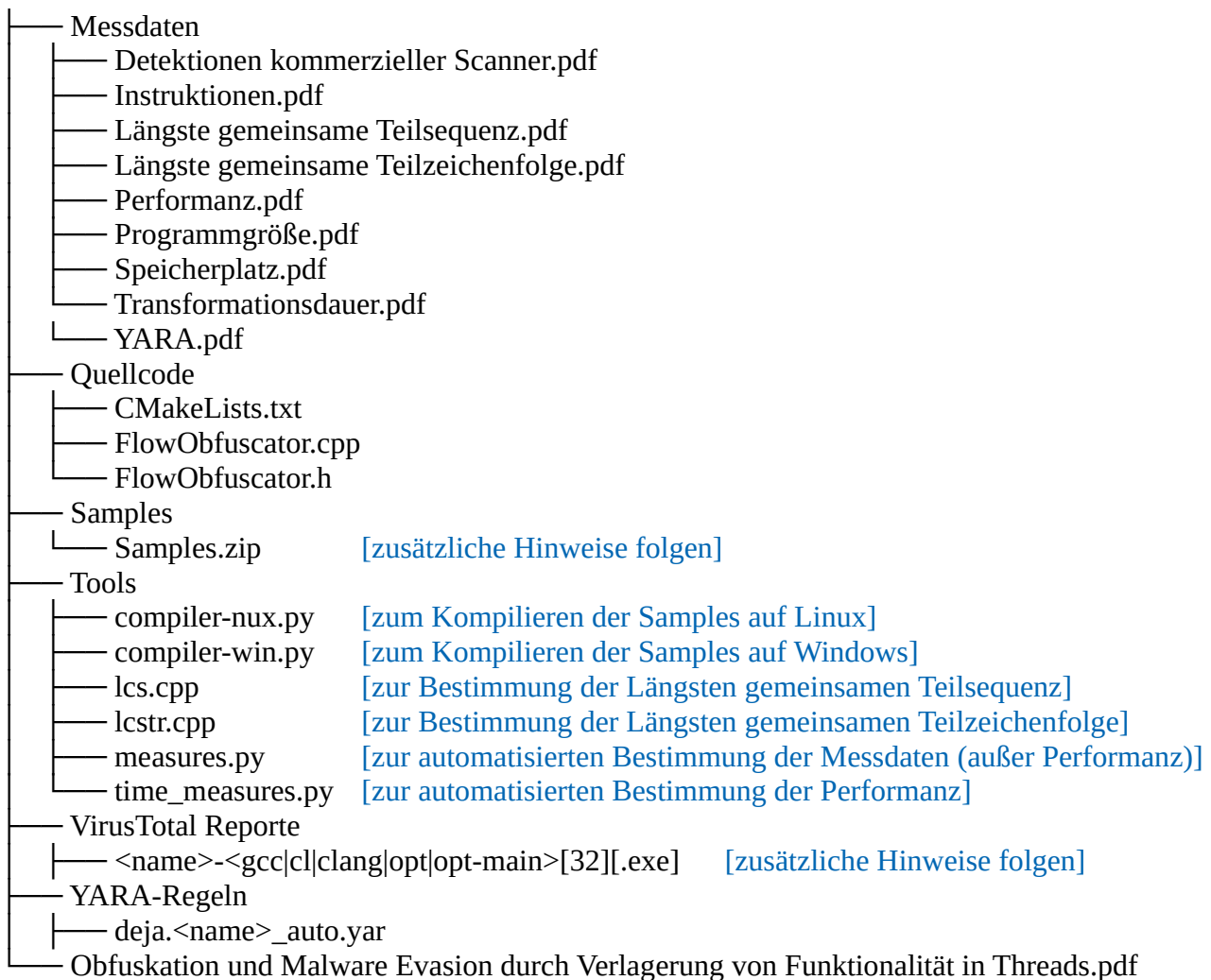


7.8 Messdaten zur Detektion durch kommerzielle Scanner

Programmname	64-bit				32-bit				Abnahme (64-bit)		Abnahme (32-bit)	
	GCC	n. obfus.	obfus.	main obfus.	GCC	n. obfus.	obfus.	main obfus.	obfus.	main obfus.	obfus.	main obfus.
Linux												
Lightraider / lightraider	9	6	5	6	7	16	9	7	1	0	7	9
BASHLITE / client	25	17	14	16	26	26	22	26	3	1	4	0
Mirai IoT Botnet / bot	26	31	28	27	27	34	31	31	3	4	3	3
Mettle / mettle	4	4	4	4	4	5	4	5	0	0	1	0
VXHeaven / Worm.FreeBSD.Block	2	2	2	2	10	11	10	10	0	0	1	1
VXHeaven / Exploit.Linux.Nhttpd	2	3	2	3	2	3	2	3	1	0	1	0
pnsniff / pnsniff	16	16	13	17	16	13	12	14	3	-1	1	-1
Mirai IoT Botnet / loader	8	7	8	7	8	8	8	8	-1	0	0	0
BASHLITE / server	4	4	4	3	3	3	3	3	0	1	0	0
VXHeaven / Virus.Unix.Adrastea	5	5	4	5	4	6	6	6	1	0	0	0
VXHeaven / Net-Worm.Linux.Slapper	2	2	2	2	2	2	2	2	0	0	0	0
VXHeaven / Exploit.Linux.Interbase	2	2	2	2	3	3	3	3	0	0	0	0
Referenz												
BBP Formel	0	0	0	0	0	0	0	0	0	0	0	0
bzip2	0	0	0	0	0	0	0	0	0	0	0	0

Programmname	64-bit				32-bit				Abnahme (64-bit)		Abnahme (32-bit)	
	CL	n. obfus.	obfus.	main obfus.	CL	n. obfus.	obfus.	main obfus.	obfus.	main obfus.	obfus.	main obfus.
Windows												
Meterpreter / msvcrt.dll	15	18	-	15	20	20	-	19	-	3	-	1
Carberp / Full.exe	-	-	-	-	47	33	17	31	-	-	16	2
Dexter / Dexter.exe	22	21	15	21	39	34	21	35	6	0	13	-1
Carberp / BotLoader2.exe	-	-	-	-	44	27	16	27	-	-	11	0
AryanRAT / AryanRAT.dll	7	7	6	7	20	20	11	12	1	0	9	8
Mimikatz / mimikatz.exe	34	30	29	30	39	33	24	31	1	0	9	2
LiquidBot / LiquidBot.exe	-	-	-	-	46	43	35	43	-	-	8	0
CyberBot / CyberBot.exe	-	-	-	-	35	40	34	40	-	-	6	0
rBot / rBot.exe	-	-	-	-	35	38	32	37	-	-	6	1
xTBot / xTBot.exe	-	-	-	-	37	34	29	35	-	-	5	-1
XOR-USB / xor-usb.exe	2	2	2	1	20	17	13	17	0	1	4	0
MyDoom.A / xproxy.dll	15	14	5	13	16	14	10	14	9	1	4	0
Carberp / CoreDll.dll	-	-	-	-	21	10	7	10	-	-	3	0
Carberp / anti_rapport.exe	-	-	-	-	28	17	15	18	-	-	2	-1
Carberp / BootkitRunBot.dll	-	-	-	-	37	9	7	9	-	-	2	0
Carberp / ddos.dll	-	-	-	-	22	6	4	7	-	-	2	-1
Grum / Grum.exe	-	-	-	-	24	24	22	22	-	-	2	2
Mimikatz / mimilove.exe	9	9	9	10	27	14	12	11	0	-1	2	3
ogw0rm / ogw0rm.exe	20	21	19	21	29	29	27	29	2	0	2	0
Meterpreter / screenshot.dll	-	-	-	-	6	8	6	6	-	-	2	2
Alina / alina.exe	20	18	15	19	40	38	37	36	3	-1	1	2
Carberp / BootkitInstallReport.exe	-	-	-	-	29	23	22	23	-	-	1	0
MyDoom.A / client.exe	2	2	2	2	6	4	3	3	0	0	1	1
ReflectiveDllInjection / ReflectiveLoader.dll	7	5	4	4	7	6	5	6	1	1	1	0
MyDoom.A / crypt1.exe	1	1	2	2	2	2	2	2	-1	-1	0	0
Dokan / dokan.dll	0	2	1	1	2	2	2	2	1	1	0	0
Meterpreter / elevator.dll	5	3	5	4	7	6	6	6	-2	-1	0	0
MyDoom.A / MyDoomA.exe	18	18	8	17	31	27	27	25	10	1	0	2
Carberp / BinToHex.exe	2	2	2	2	6	4	5	6	0	0	-1	-2
MyDoom.A / cleanpe.exe	4	3	2	2	2	3	4	4	1	1	-1	-1
Polymorphic Virus / encrypt.exe	1	1	1	0	1	1	2	2	0	1	-1	-1
MyDoom.A / bin2c.exe	1	1	2	2	2	2	4	4	-1	-1	-2	-2
ReflectiveDllInjection / Inject.exe	5	5	5	4	9	8	10	8	0	1	-2	0
Mimikatz / mimilib.dll	26	22	20	21	26	19	21	21	2	1	-2	-2
Mimikatz / mimispool.dll	14	14	13	14	14	13	15	12	1	0	-2	1
MyDoom.A / rot13.exe	15	2	1	2	2	1	3	1	1	0	-2	0
XOR-USB / xor.exe	1	1	1	1	3	2	6	4	0	0	-4	-2
Dokan / dokan_mount.exe	2	2	3	3	9	7	19	7	-1	-1	-12	0
Referenz												
BBP Formel	1	1	1	0	7	20	4	20	0	1	16	0
bzip2	1	1	1	12	2	3	3	3	0	-11	0	0

Gliederung der beigelegten CD



Hinweise zu den Samples: Da die Samples hauptsächlich aus Malware bestehen und Schwierigkeiten beim Einlegen der CD durch Malware Scanner entstehen können, wurden die Samples in einem ZIP-Archiv komprimiert und mit dem Passwort „infected“ versehen. Die Syntax der Namen der im Archiv enthaltenen Dateien entspricht der Syntax bei den VirusTotal Reporten.

Hinweise zum Dateinamen der Samples und den VirusTotal Reporten:

- **<name>**: Name des ausführbaren Datei. Eine Zuordnung zum Projekt kann über die Messtabelle in „Detektionen kommerzieller Scanner.pdf“ erfolgen.
- **<gcc|cl|clang|opt|opt-main>**: Die jeweilige Variante.
 - **gcc**: nicht obfuskiert mit dem GCC Compiler
 - **cl**: nicht obfuskiert mit dem MSVC Compiler
 - **clang**: nicht obfuskiert mit dem Clang Compiler
 - **opt**: vollständig obfuskiert
 - **opt-main**: nur *main*-Funktion obfuskiert
- **[32]**: Die 32-bit Variante (optional).
- **[.exe]**: Eine PE-Datei (anstatt einer ELF-Datei).

Abkürzungsverzeichnis

API	Application Programming Interface
ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
ELF	Executable and Linking Format
GPU	Graphics Processing Unit
IR	Intermediate Representation
LCS	Longest Common Subsequence
LOC	Lines Of Code
MIST	Malware Instruction Set
MSVC	Microsoft Visual C++
OS	Operating System
PC	Personal Computer
PE	Portable Executable
SVM	Support Vector Machine

Abbildungsverzeichnis

Abbildung 1: Ghidras Benutzeroberfläche.....	6
Abbildung 2: Schematische und illustrative Darstellung des Obfuskationsalgorithmus.....	17
Abbildung 3: Zunahme der Programmgröße.....	32
Abbildung 4: Zunahme der Instruktionen.....	36
Abbildung 5: Koordinatensystem zur Einordnung der Widerstandsfähigkeit der Thread-basierten Obfuskation [2].....	37
Abbildung 6: Zunahme der Übersetzungszeit bei Anwendung der Obfuskation.....	40
Abbildung 7: Gegenüberstellung der Zunahme des dynamischen und statischen Speichers.....	41
Abbildung 8: Zunahme des statischen Speichers unter Berücksichtigung der Ordnung der Zunahme des dynamischen Speichers.....	41
Abbildung 9: Gegenüberstellung BBP Formel und bzip2 hinsichtlich der Zunahme der Ausführungsdauer.....	42
Abbildung 10: Spannungsdreieck der drei vorgestellten Obfuskationsmetriken.....	45
Abbildung 11: Gegenüberstellung des obfuskierten und nicht obfuskierten Kontrollflussgraphen...	48
Abbildung 12: Ausschnitt des Funktionsaufrufgraphen der main-Funktion des nicht obfuskierten Programms.....	48

Codeausschnittsverzeichnis

Codeausschnitt 1: Nicht obfuskierte Funktion.....	12
Codeausschnitt 2: Obfuskierte Funktion.....	13
Codeausschnitt 3: Simpler Beispielcode in C.....	20
Codeausschnitt 4: Gegenüberstellung der auf Basis von Codeausschnitt 3 generierten LLVM IR...	20
Codeausschnitt 5: Gegenüberstellung des auf Basis von Codeausschnitt 4 generierten Maschinencodes in Assemblerdarstellung (AT&T Syntax).....	21
Codeausschnitt 6: Kommandozeilenbefehle zur Prüfung der Korrektheit des obfuskierten Programms.....	29
Codeausschnitt 7: Testergebnisse des Coreutils Softwarepakets.....	30
Codeausschnitt 8: Erster Eindruck zwischen obfuskiertem und nicht obfuskiertem main-Funktion....	47
Codeausschnitt 9: Ausgabe des Decompilers zur Anfangsfunktionalität der main-Funktion.....	49
Codeausschnitt 10: Gegenüberstellung des Stacktraces beim obfuskierten und nicht obfuskierten Programm.....	51
Codeausschnitt 11: TLSHs von einigen Samples des bzip2-Programms.....	52

Tabellenverzeichnis

Tabelle 1: Betriebssystemabhängige Funktionen und Typen für die Realisierung der Obfuskation..	19
Tabelle 2: Verhältnis zwischen der BBP Formel und bzip2 hinsichtlich der Performanz.....	43
Tabelle 3: Detektionen durch die vom YARA-Signator generierten YARA-Regeln.....	54
Tabelle 4: Anzahl der Detektionen durch VirusTotal.....	61

Literaturverzeichnis

- [1] Banescu, Sebastian & Pretschner, Alexander. (2017). A Tutorial on Software Obfuscation. *Advances in Computers*, vol. 108, pp. 283–353, Elsevier. 10.1016/bs.adcom.2017.09.004.
- [2] Collberg, Christian & Thomborson, Clark & Low, Douglas. (1997). A Taxonomy of Obfuscating Transformations.
- [3] Watt, David & Findlay, William. (2004). Programming language design concepts.
- [4] Intel. (2021). Intel 64 and IA-32 Architectures Software Developer's Manual.
- [5] Markgraf, Daniel. (2018). Reverse Engineering.
<https://wirtschaftslexikon.gabler.de/definition/reverse-engineering-45260/version-268557>.
Letzter Zugriff 06.05.2021.
- [6] Yakdan, Khaled & Dechand, Sergej & Gerhards-Padilla, Elmar & Smith, Matthew. (2016). Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. 2016 IEEE Symposium on Security and Privacy (SP), pp. 158-177. 10.1109/SP.2016.18.
- [7] Ghidra. <https://ghidra-sre.org/>. Letzter Zugriff 18.04.2021.
- [8] Eilam, E. (2005). Reversing: Secrets of Reverse Engineering.
- [9] Choi, Jusop & Shin, Dongsoon & Kim, Hyoungshick & Seotis, Jason & Hong, Jin. (2019). AMVG: Adaptive Malware Variant Generation Framework Using Machine Learning. 2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 246-24609. 10.1109/PRDC47002.2019.00055.
- [10] Wenzl, Matthias & Merzdovnik, Georg & Ullrich, Johanna & Weippl, Edgar. (2019). From Hack to Elaborate Technique—A Survey on Binary Rewriting. *ACM Computing Surveys* 52, pp 1-37. 10.1145/3316415.
- [11] Bruschi, Danilo & Martignoni, Lorenzo & Monga, Mattia. (2006). Using Code Normalization for Fighting Self-Mutating Malware.
- [12] Bundesamt für Sicherheit in der Informationstechnik.
https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Informationen-und-Empfehlungen/Cyber-Sicherheitsempfehlungen/Virenschutz-Firewall/Virenschutzprogramme/virenschutzprogramme_node.html. Letzter Zugriff 06.05.2021.
- [13] Christodorescu, Mihai & Jha, Somesh & Maughan, Douglas & Song, Dawn & Wang, Cliff. (2006). *Malware Detection (Advances in Information Security)*. Springer-Verlag, Berlin, Heidelberg.
- [14] Damodaran, Anusha & Di Troia, Fabio & Visaggio, Corrado Aaron & Austin, Thomas & Stamp, Mark. (2017). A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques* 13, pp. 1-12. 10.1007/s11416-015-0261-z.

- [15] Trend Micro. TLSH – Trend Micro Locality Sensitive Hash. <https://github.com/trendmicro/tlsh>. Letzter Zugriff 18.04.2021.
- [16] Upchurch, Jason & Zhou, Xiaobo. (2015). Variant: a malware similarity testing framework. 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 31-39. 10.1109/MALWARE.2015.7413682.
- [17] VirusTotal. Yara. <https://virustotal.github.io/yara/>. Letzter Zugriff 18.04.2021.
- [18] Bonfante, Guillaume & Kaczmarek, Matthieu & Marion, Jean-Yves. (2007). Control Flow Graphs as Malware Signatures.
- [19] Bruschi, Danilo & Martignoni, Lorenzo & Monga, Mattia. (2006). Detecting Self-mutating Malware Using Control-Flow Graph Matching. Detection of Intrusions and Malware & Vulnerability Assessment. DIMVA 2006. 129-143. 10.1007/11790754_8.
- [20] Champion, Marco & Dalla Preda, Mila & Giacobazzi, Roberto. (2021). Learning metamorphic malware signatures from samples. Journal of Computer Virology and Hacking Techniques, pp. 1-17. 10.1007/s11416-021-00377-z.
- [21] Eskandari, Mojtaba & Hashemi, Sattar. (2012). A graph mining approach for detecting unknown malwares. Journal of Visual Languages & Computing 23, pp. 154–162. 10.1016/j.jvlc.2012.02.002.
- [22] Adkins, Francis & Jones, Luke & Carlisle, Martin & Upchurch, Jason. (2013). Heuristic malware detection via basic block comparison. 2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE), pp. 11-18. 10.1109/MALWARE.2013.6703680.
- [23] Cohen, Fred. (1990). A Short Course on Computer Viruses. Computers & Security - COMP-SEC 8.
- [24] Microsoft. (2021). Über die Cloud bereitgestellten Netzwerkschutz aktivieren. <https://docs.microsoft.com/de-de/microsoft-365/security/defender-endpoint/enable-cloud-protection-microsoft-defender-antivirus?view=o365-worldwide>. Letzter Zugriff 01.08.2021.
- [25] Cuckoo. <https://cuckoosandbox.org/>. Letzter Zugriff 18.04.2021.
- [26] Raff, Edward & Barker, Jon & Sylvester, Jared & Brandon, Robert & Catanzaro, Bryan & Nicholas, Charles. (2017). Malware Detection by Eating a Whole EXE. AAAI Workshops.
- [27] Rieck, Konrad & Trinius, Philipp & Willems, Carsten & Holz, Thorsten. (2011). Automatic analysis of malware behavior using machine learning. Journal of Computer Security 19, pp. 639-668. 10.3233/JCS-2010-0410.
- [28] Ahmed, Faraz & Hameed, Haider & Shafiq, M. & Farooq, Muddassar. (2009). Using spatio-temporal information in API calls with machine learning algorithms for malware detection. AISEC '09: Proceedings of the 2nd ACM workshop on Security and artificial intelligence, pp. 55-62. 10.1145/1654988.1655003.
- [29] Firdausi, Ivan & Lim, Charles & Erwin, Alva & Nugroho, Anto. (2010). Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection. 2010 Second Internati-

- onal Conference on Advances in Computing, Control, and Telecommunication Technologies, pp. 201-203. 10.1109/ACT.2010.33.
- [30] Souri, Alireza & Hosseini, Rahil. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences* 8, pp. 1-22. 10.1186/s13673-018-0125-x.
 - [31] Kaspersky. Machine Learning Methods for Malware Detection. (2020). <https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf>. Letzter Zugriff 18.04.2021.
 - [32] Sophos. Machine Learning: How to Build a Better Threat Detection Model. (2017). <https://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/machine-learning-how-to-build-a-better-threat-detection-model.pdf>. Letzter Zugriff 18.04.2021.
 - [33] Zhang, Qinghua & Reeves, D.s. (2007). MetaAware: Identifying Metamorphic Malware. *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 411-420. 10.1109/ACSAC.2007.9.
 - [34] Wong, Wing & Stamp, Mark. (2006). Hunting for metamorphic engines. *Journal in Computer Virology* 2, pp. 211-229. 10.1007/s11416-006-0028-7.
 - [35] Marpaung, Jonathan & Sain, Mangal & Lee, Hoon-Jae. (2012). Survey on malware evasion techniques: State of the art and challenges. *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pp. 744-749.
 - [36] Afianian, Amir & Niksefat, Salman & Sadeghiyan, Babak & Baptiste, David. (2019). Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Computing Surveys* 52, pp. 1-28. 10.1145/3365001.
 - [37] UPX. <https://upx.github.io/>. Letzter Zugriff 18.04.2021.
 - [38] Reece, Alex. (2013). Introduction to return oriented programming (ROP). <https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html>. Letzter Zugriff 20.05.2021.
 - [39] Demetrio, Luca & Biggio, Battista & Lagorio, Giovanni & Roli, Fabio & Armando, Alessandro. (2019). Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries.
 - [40] Rosenberg, Ishai & Gudes, Ehud. (2016). Bypassing system calls-based intrusion detection systems. *Concurrency and Computation: Practice and Experience*. 29. 10.1002/cpe.4023.
 - [41] Li, Xufang & Loh, Peter & Tan, Freddy. (2011). Mechanisms of Polymorphic and Metamorphic Viruses. *2011 European Intelligence and Security Informatics Conference*, pp. 149-154. 10.1109/EISIC.2011.77.
 - [42] Borello, Jean-Marie & Mé, Ludovic. (2008). Code Obfuscation Techniques for Metamorphic Viruses. *Journal in Computer Virology* 4, pp. 211-220. 10.1007/s11416-008-0084-2.

- [43] Murad, Khurram & Shirazi, Syed & Zikria, Yousaf & Ikram, Nassar. (2010). Evading Virus Detection Using Code Obfuscation. Future Generation Information Technology, pp. 394-401. 10.1007/978-3-642-17569-5_39.
- [44] Payer, Mathias. (2014). Embracing the new threat: towards automatically, self-diversifying malware.
- [45] Junod, Pascal & Rinaldini, Julien & Wehrli, Johan & Michielin, Julie. (2015). Obfuscator-LLVM -- Software Protection for the Masses. 2015 IEEE/ACM 1st International Workshop on Software Protection, pp. 3-9. 10.1109/SPRO.2015.10.
- [46] Larsen, Per & Homescu, Andrei & Brunthaler, Stefan & Franz, Michael. (2014). SoK: Automated Software Diversity. 2014 IEEE Symposium on Security and Privacy, pp. 276-291. 10.1109/SP.2014.25.
- [47] Veerappan, Chandra Sekar & Keong, Peter & Tang, Zhaohui & Tan, Forest. (2018). Taxonomy on malware evasion countermeasures techniques. 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), pp. 558-563. 10.1109/WF-IoT.2018.8355202.
- [48] Yuval tisf Nativ. TheZoo – A Live Malware Repository. <https://github.com/ytisf/theZoo>. Letzter Zugriff 18.04.2021.
- [49] Ma, Weiqin & Duan, Pu & Liu, Sanmin & Gu, Guofei & Liu, Jyh-charn. (2011). Shadow attacks: Automatically evading system-call-behavior based malware detection. Journal in Computer Virology 8, pp. 1-13. 10.1007/s11416-011-0157-5.
- [50] Omar, Rasha & El-Mahdy, Ahmed & Rohou, Erven. (2013). Thread-Based Obfuscation through Control-Flow Mangling.
- [51] LLVM- The LLVM Compiler Infrastructure. <https://llvm.org/>. Letzter Zugriff 18.04.2021.
- [52] Statista. (2021). Marktanteile der führenden Betriebssysteme weltweit im Oktober 2020. <https://de.statista.com/statistik/daten/studie/828610/umfrage/marktanteile-der-fuehrenden-betriebssystemversionen-weltweit/>. Letzter Zugriff 20.05.2021.
- [53] Andrzej Warzynski. How to create pass independently on Windows. (2020). <https://llvm.discourse.group/t/how-to-create-pass-independently-on-windows/474>. Letzter Zugriff 18.04.2021.
- [54] LLVM. Clang: a C language family frontend for LLVM. <https://clang.llvm.org/>. Letzter Zugriff 18.04.2021.
- [55] Statista. (2021). Die beliebtesten Programmiersprachen weltweit laut PYPL-Index im März 2021. <https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebteste-programmiersprachen-weltweit-laut-pypl-index/>. Letzter Zugriff 20.05.2021.
- [56] Sutter, Herb & Alexandrescu, Andrei. (2004). C++ Coding Standards.
- [57] LLVM. Exception Handling in LLVM. <https://llvm.org/docs/ExceptionHandling.html>. Letzter Zugriff 18.04.2021.

- [58] Linux man page. pthread_mutex_lock(3). (2003).
https://linux.die.net/man/3/pthread_mutex_lock. Letzter Zugriff 18.04.2021.
- [59] Linux manual page. pthread_spin_lock(3). (2021).
https://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html. Letzter Zugriff 18.04.2021.
- [60] Linux manual page. (2021). pthread_cancel(3).
https://man7.org/linux/man-pages/man3/pthread_cancel.3.html. Letzter Zugriff 05.07.2021.
- [61] Microsoft. (2018). TerminateThread function.
<https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread>. Letzter Zugriff 05.07.2021.
- [62] Microsoft. (2018). Slim Reader/Writer (SRW) Locks.
<https://docs.microsoft.com/en-us/windows/win32/sync/slim-reader-writer--srw--locks>. Letzter Zugriff 12.07.2021.
- [63] Trail of Bits. McSema. <https://github.com/lifting-bits/mcsema>. Letzter Zugriff 18.04.2021.
- [64] Hex Rays. IDA Pro. <https://www.hex-rays.com/products/ida/>. Letzter Zugriff 18.04.2021.
- [65] Petr Zemek. Compiling back LLVM IR. <https://github.com/avast/retdec/issues/45>. Letzter Zugriff 18.04.2021.
- [66] mk-mode. C++ source code to compute pi with BBP formula.
<https://gist.github.com/komasaru/9e418eb666ab649ef589>. Letzter Zugriff 18.04.2021.
- [67] Julian Seward. bzip2. <https://github.com/opencor/bzip2>. Letzter Zugriff 18.04.2021.
- [68] GNU. Coreutils – GNU core utilities. (2020). <https://www.gnu.org/software/coreutils/>. Letzter Zugriff 28.06.2021.
- [69] Moser, Andreas & Kruegel, Christopher & Kirda, Engin. (2008). Limits of Static Analysis for Malware Detection. Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421-430. 10.1109/ACSAC.2007.21.
- [70] Ding, Fei. (2017). IoT Malware. <https://github.com/ifding/iot-malware>. Letzter Zugriff 03.05.2021.
- [71] Rapid7. Mettle. <https://github.com/rapid7/mettle>. Letzter Zugriff 04.06.2021.
- [72] Karnik, Abhishek & Goswami, Suchandra. (2007). Detecting Obfuscated Viruses Using Cosine Similarity Analysis. First Asia International Conference on Modelling & Simulation (AMS'07), pp. 165-170. 10.1109/AMS.2007.31.
- [73] Shabtai, Asaf & Moskovitch, Robert & Feher, Clint & Dolev, Shlomi & Elovici, Yuval. (2012). Detecting unknown malicious code by applying classification techniques on OpCode patterns. Security Informatics 1, pp. 1-22. 10.1186/2190-8532-1-1.
- [74] Kolter, Jeremy & Maloof, Marcus. (2006). Learning to Detect and Classify Malicious Executables in the Wild. Journal of Machine Learning Research 7, pp. 2721-2744.

- [75] Graham Cluley. VX Heaven, old-school virus-writing website, returns from the dead after police raid. <https://grahamcluley.com/vx-heaven/>. Letzter Zugriff 28.05.2021.
- [76] VX Heaven. VX Heaven Virus Collection 2010-05-18. <https://academictorrents.com/details/34ebe49a48aa532deb9c0dd08a08a017aa04d810/tech>. Letzter Zugriff 18.04.2021.
- [77] Rapid7. Windows Native C meterpreter. <https://github.com/rapid7/metasploit-payloads/tree/master/c/meterpreter>. Letzter Zugriff 28.06.2021.
- [78] Benjamin Delpy. mimikatz. <https://github.com/gentilkiwi/mimikatz>. Letzter Zugriff 28.06.2021.
- [79] Nick Allen, Tyler Falther, Laine Rumreich, Collin Thomas, Nathan Simpson, Patrick Walter. Polymorphic Virus & Countermeasures. (2019). <https://github.com/LaineRumreich/PolymorphicVirus>. Letzter Zugriff 28.06.2021.
- [80] Stephen Fewer. ReflectiveDLLInjection. (2013). <https://github.com/stephenfewer/ReflectiveDLLInjection>. Letzter Zugriff 28.06.2021.
- [81] Linux manual page. (2021). diff(1). <https://man7.org/linux/man-pages/man1/diff.1.html>. Letzter Zugriff 28.06.2021.
- [82] Pádraig Brady. (2017). How the GNU coreutils are tested. <http://www.pixelbeat.org/docs/coreutils-testing.html>. Letzter Zugriff 14.07.2021.
- [83] Linux manual page. (2021). signal(2). <https://man7.org/linux/man-pages/man2/signal.2.html>. Letzter Zugriff 05.07.2021.
- [84] Omar, Rasha & El-Mahdy, Ahmed & Rohou, Erven. (2014). Arbitrary Control-Flow Embedding into Multiple Threads for Obfuscation: A Preliminary Complexity and Performance Analysis. SCC '14: Proceedings of the 2nd international workshop on Security in cloud computing, pp. 51-58. 10.1145/2600075.2600080.
- [85] Gusfield, Dan. (1997). Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge, UK: Cambridge University Press.
- [86] Coco, Moreno & Keller, Frank. (2012). Scan Patterns Predict Sentence Production in the Cross-Modal Processing of Visual Scenes. Cognitive science, vol. 36, pp. 1204-1023. 10.1111/j.1551-6709.2012.01246.x.
- [87] Haq, Irfan Ul & Caballero, Juan. (2021). A Survey of Binary Code Similarity. ACM Computing Surveys, vol. 54, pp. 1-38. 10.1145/3446371.
- [88] Linux Kernel Organization. Perf: Linux profiling with performance counters. (2020). https://perf.wiki.kernel.org/index.php/Main_Page. Letzter Zugriff 20.07.2021.
- [89] Denis Bakhvalov. (2019). How to get consistent results when benchmarking on Linux?“. [https://easyp erf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux#4-set-cpu-affinity](https://easyp perf.net/blog/2019/08/02/Perf-measurement-environment-on-Linux#4-set-cpu-affinity). Letzter Zugriff 22.07.2021.

- [90] Linux manual page. (2019). time(1). <https://man7.org/linux/man-pages/man1/time.1.html>. Letzter Zugriff 23.07.2021.
- [91] GNU. GDB: The GNU Project Debugger. <https://www.gnu.org/software/gdb/>. Letzter Zugriff 18.04.2021.
- [92] x64dbg. <https://x64dbg.com/>. Letzter Zugriff 18.04.2021.
- [93] GNU. GCC, the GNU Compiler Collection. <https://gcc.gnu.org/>. Letzter Zugriff 18.04.2021.
- [94] VirusTotal. Contributors. <https://support.virustotal.com/hc/en-us/articles/115002146809-Contributors>. Letzter Zugriff 10.08.2021.
- [95] Bilstein, Felix & Plohmann, Daniel. (2019). YARA-Signator: Automated Generation of Code-based YARA Rules. The Journal on Cybercrime & Digital Investigations, vol. 5, no. 1. Botconf 2019 Proceedings. 10.18464/cybin.v5i1.24.
- [96] Sulaiman, A. & Ramamoorthy, K. & Mukkamala, Srinivas & Sung, Andrew. (2005). Malware examiner using disassembled code (MEDIC). Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, pp. 428-429. 10.1109/IAW.2005.1495985.
- [97] Oracle. VirtualBox. <https://www.virtualbox.org/>. Letzter Zugriff 18.04.2021.
- [98] Thomas Hungenberg & Matthias Eckert. (2020). InetSim: Internet Services Simulation Suite. <https://www.inetsim.org/>. Letzter Zugriff 29.07.2021.
- [99] Qiao, Yong & He, Jie & Yang, Yuexiang & Ji, Lin. (2013). Analyzing Malware by Abstracting the Frequent Itemsets in API Call Sequences. 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, pp. 265-270. 10.1109/TrustCom.2013.36.
- [100] Dai, Jianyong & Joohan, Lee. (2009). Efficient Virus Detection Using Dynamic Instruction Sequences. Journal of Computers 4, pp. 405-414. 10.4304/jcp.4.5.405-414.
- [101] Wendelin, Claes Adam. (2017). Malware Detection in Secured Systems. Seminar Future Internet SS2017. 10.2313/NET-2017-09-1_11.
- [102] Sorokin, Ivan. (2011). Comparing files using structural entropy. Journal in Computer Virology 7, pp. 259-265. 10.1007/s11416-011-0153-9.
- [103] Alexander H. Liu. (2018). MalConv-Pytorch. <https://github.com/Alexander-H-Liu/MalConv-Pytorch>. Letzter Zugriff 18.07.2021.
- [104] Kondrad Rieck. Malheur Dataset. <https://www.sec.cs.tu-bs.de/data/malheur/>. Letzter Zugriff 29.07.2021.
- [105] VirusTotal. <https://www.virustotal.com/>. Letzter Zugriff 18.04.2021.
- [106] Sharif, Mahmood & Lucas, Keane & Bauer, Lujo & Reiter, Michael & Shintre, Saurabh. (2019). Optimization-Guided Binary Diversification to Mislead Neural Networks for Malware Detection.

- [107] Castro, Raphael & Schmitt, Corinna & Rodosek, Gabi. (2019). ARMED: How Automatic Malware Modifications Can Evade Static Detection?. 2019 5th International Conference on Information Management (ICIM), pp. 20-27. 10.1109/INFOMAN.2019.8714698.
- [108] Demetrio, Luca & Biggio, Battista & Lagorio, Giovanni & Roli, Fabio & Armando, Alessandro. (2020). Functionality-preserving Black-box Optimization of Adversarial Windows Malware. IEEE Transactions on Information Forensics and Security. 10.1109/TIFS.2021.3082330.
- [109] Rossow, Christian & Dietrich, Christian & Grier, Chris & Kreibich, Christian & Paxson, Vern & Pohlmann, Norbert & Bos, Herbert & van Steen, Maarten. (2012). Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. 2012 IEEE Symposium on Security and Privacy, pp. 65-79. 10.1109/SP.2012.14.
- [110] Hu, Weiwei & Tan, Ying. (2017). Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN.
- [111] Pavithran, Jithin & Patnaik, Milan & Rebeiro, Chester. (2019). D-TIME: Distributed Thresholdless Independent Malware Execution for Runtime Obfuscation. WOOT @ USENIX Security Symposium.
- [112] Rosenberg, Ishai & Shabtai, Asaf & Rokach, Lior & Elovici, Yuval. (2018). Generic Black-Box End-to-End Attack against RNNs and Other API Calls Based Malware Classifiers. Research in Attacks, Intrusions, and Defenses. 10.1007/978-3-030-00470-5_23.
- [113] Zhou, Ruuyu. (2018). Guided Automatic Binary Parallelisation. 10.17863/CAM.21890.
- [114] Wikipedia. Pointer analysis. https://en.wikipedia.org/wiki/Pointer_analysis. Letzter Zugriff 18.04.2021.
- [115] Wikipedia. Längster Pfad. https://de.wikipedia.org/wiki/L%C3%A4ngster_Pfad. Letzter Zugriff 18.04.2021.
- [116] Linux manual page. Getrlimit(2). (2021). <https://man7.org/linux/man-pages/man2/setrlimit.2.html>. Letzter Zugriff 19.05.2021.
- [117] LLVM. GlobalOpt.cpp. <https://github.com/llvm/llvm-project/blob/llvmorg-11.1.0/llvm/lib/Transforms/IPO/GlobalOpt.cpp>. Letzter Zugriff 18.04.2021.