Christian-Albrechts-Universität zu Kiel

Technische Fakultät

Masterarbeit in Informatik

Maschinencode-Obfuscation als Schutz vor Reverse Engineering

von Folke Will

PAX OPTIMA RERUM

Masterarbeit in Informatik

Maschinencode-Obfuscation als Schutz vor Reverse Engineering

Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion

Institut für Informatik

Christian-Albrechts-Universität zu Kiel

vorgelegt von

Folke Will

Betreut von:	Prof. Dr. Michael Hanus
	und Fabian Skrlac
Datum:	24. September 2014

Zusammenfassung

In dieser Arbeit wird erläutert, wie Software bereits während des Kompiliervorgangs gegen Reverse-Engineering-Angriffe geschützt werden kann. Bisherige Ansätze führen eine spezielle Stufe zwischen Kompilieren und Linken ein, was zu einigen Nachteilen führt. Es werden daher zunächst bekannte Verfahren direkt in einen Compiler integriert, um die Effektivität dieses Ansatzes zu evaluieren. Anschließend werden eigene Verfahren vorgestellt, die durch diese Integration erst möglich werden. In einer abschließenden Diskussion werden die Vor- und Nachteile des neuen Ansatzes analysiert.

Abstract

This thesis describes an approach to protect software against reverse engineering attacks by integrating counter measures directly into a compiler. Previous approaches usually introduced an external stage between compilation and linking, leading to several disadvantages. In order to test the compiler-driven approach, previous methods of protection will be integrated into a compiler. Afterwards, some new techniques that are made possible by the compiler integration will be presented. Finally, the new approach will be discussed and its advantages and disadvantages will be analyzed.

Eidesstattliche Versicherung der Selbständigkeit

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegt wurde.

Folke Will – Kiel, den 24. September 2014

Inhaltsverzeichnis

1 Einleitung

1.1 Einführung

Bei der Entwicklung von Software wird oft unbewusst angenommen, dass durch die Übersetzung eines in einer Hochsprache geschriebenen Quellcodes ein ausführbares Maschinenprogramm entsteht, aus dem keine Informationen über Details der Implementierung entnommen werden können. Bei dieser Annahme wird vernachlässigt, dass ausführbare Maschinenprogramme letztlich Maschinencodeanweisungen enthalten, die vom Betriebssystem geladen und vom Prozessor ausgeführt werden. Da die Syntax und Semantik dieser Anweisungen bekannt sind und die Details über den Aufbau der Dateiformate für Maschinenprogramme ebenfalls bekannt sind, kann ein kompiliertes Programm nicht nur ausgeführt, sondern auch inhaltlich anhand des Maschinencodes analysiert werden. Darüber hinaus können Programme auch manipuliert werden, indem die Maschinencodeanweisungen geändert werden. Sowohl die Analyse als auch die Manipulation sind ohne Kenntnis des Hochsprachencodes möglich, aus dem das Programm erzeugt wurde.

Eine solche Analyse oder Manipulation ist insbesondere bei kommerziellen Programmen problematisch, die als Maschinenprogramm ohne Quellcode verkauft werden. Denn hier hat der Entwickler des Programms üblicherweise ein Interesse daran, dass ein etwaiger Kopierschutz des Programms nicht manipuliert wird. Im Kontext von Industriespionage ist es darüber hinaus möglich, Software-Produkte von Mitbewerbern zu analysieren, um die dort verwendeten Algorithmen zu rekonstruieren und in eigenen Produkten zu verwenden. Insbesondere Methoden zur Implementierung eines Kopierschutzes werden oft analysiert und manipuliert: es gibt Gruppierungen mit anonymen Teilnehmern, die sich darauf spezialisiert haben, den Kopierschutz von kommerzieller Software zu umgehen und die Ergebnisse öffentlich verfügbar zu machen. Zwischen diesen Gruppen herrscht ein reger Wettbewerb bezüglich der Geschwindigkeit der Analysen. Da es diese Gruppen bereits seit den 1980er Jahren gibt, ist die Spezialisierung mittlerweile so weit fortgeschritten, dass für viele neue Produkte noch am Tag der Veröffentlichung eine Methode zur Umgehung des Kopierschutzes entwickelt und veröffentlicht wird [KPW12]. Analysen dieser Art fallen unter den Begriff des *Reverse Engineering*. Darunter werden in diesem Kontext allgemein Methoden verstanden, die zur Analyse oder Manipulation eines Maschinenprogramms genutzt werden können, ohne dass dessen Hochsprachencode bekannt ist. Im Rahmen dieser Arbeit werden solche Methoden als *Angriffe* bezeichnet, die von einem nicht näher definierten *Angreifer* erfolgen.

Um Software vor Angriffen dieser Art zu schützen, kann *Obfuscation* als Gegenmaßnahme eingesetzt werden. Obfuscation ist eine Technik, bei der Maschinencode auf eine solche Weise verschleiert wird, dass eine Analyse im Rahmen von Reverse Engineering schwieriger ist. Theoretische Ergebnisse haben gezeigt, dass keine Verschleierung existieren kann, die alle Programme automatisiert vor Reverse Engineering schützen kann [Bar+12]. Dieses Ergebnis ist auch praktisch nachvollziehbar: Auch ein transformiertes Programm besteht aus Maschinencodeanweisungen, die bei der Ausführung durch den Prozessor dasselbe Ergebnis wie das ursprüngliche Programm berechnen – und diese Maschinencodeanweisungen können ebenso analysiert werden. Obfuscation kann Reverse Engineering also nur erschweren, aber

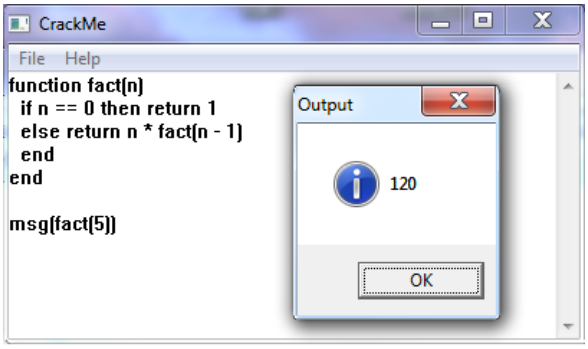


Abbildung 1: Die Beispielanwendung implementiert einen Interpreter für die Programmiersprache Lua. In das Textfeld kann Lua-Code eingegeben werden, der über das Menü ausgeführt werden kann. Der Code ist in der Länge beschränkt: Bei Quellcodes, die eine bestimmte Länge überschreiten, wird ein Hinweis ausgegeben, dass in der Testversion nur kurze Quellcodes erlaubt sind. Über einen weiteren Menüpunkt kann die Anwendung durch Eingabe von Anwendername und Seriennummer registriert werden, wodurch die Beschränkung der Eingabelänge aufgehoben wird.

nicht verhindern. „Schwierig“ bedeutet hierbei, dass der Aufwand des Angreifers steigt, um das Verfahren anzugreifen und das Programm analysieren zu können.

In dieser Arbeit werden zunächst grundlegende Techniken des Reverse Engineering eingeführt, um die Möglichkeiten des Angreifers zu verstehen. Dazu wird ein beispielhaftes Programm untersucht (siehe Abbildung 1), das einen Kopierschutz in Form einer personalisierten Seriennummer implementiert. Es wird gezeigt, dass dieser Schutz mittels Reverse Engineering umgangen und in einem weiteren Schritt sogar eine gültige Seriennummer erzeugt werden kann. Danach werden bestehende Techniken zur Verschleierung im Hinblick auf ihre Angreifbarkeit überprüft. Anschließend wird untersucht, ob sich bestehende Verfahren verbessern lassen oder neue entwickelt werden können, wenn der Verschleierungsvorgang in einen Übersetzer integriert wird. Abschließend wird die Beispielanwendung hinsichtlich der Erschwerung solcher Angriffe verbessert.

1.2 Beschreibung der Zielplattform

Da die Umsetzung von Verschleierungstechniken stark an die Architektur des Zielsystems der Maschinenprogramme gebunden ist, wird die für diese Arbeit gewählte Rechnerarchitektur kurz beschrieben. Als Plattform wird in dieser Arbeit Microsoft Windows in der Variante für x86-CPUs verwendet, da diese Plattform für die meisten kommerziellen Programme verwendet wird, bei denen der Quellcode nicht verfügbar ist. Prinzipiell funktionieren die vorgestellten Verschleierungen auch unter Betriebssystemen wie Linux, allerdings werden dort ohnehin die meisten Programme inklusive Quellcode veröffentlicht.

Die Architektur des x86-Prozessors ist von Intel gut dokumentiert [Int14] und soll hier kurz erläutert werden, um das Verständnis der Arbeit zu erleichtern. Ein x86-kompatibler

Register	32 Bit	Unterste 16 Bit	Davon obere 8 Bit	Unterste 8 Bit
0	EAX	AX	AH	AL
1	ECX	CX	CH	CL
2	EDX	DX	DH	DL
3	EBX	BX	BH	BL
4	ESP	SP	-	-
5	EBP	BP	-	-
6	ESI	SI	-	-
7	EDI	DI	-	-

Tabelle 2: Die Tabelle zeigt die verfügbaren Register der x86-Familie. Die unteren Bits der Allzweckregister sind über verschiedene Bezeichnet ansprechbar – eine Manipulation des Registers AL ändert beispielsweise die untersten 8 Bits des Registers EAX. Quelle: nach [Int14, Kap. 3-12].

Prozessor besitzt 8 Basisregister (vgl. [Int14, Kap. 3-10]), die in den ersten Generationen des Intel 8086-Prozessors für fest definierte Zwecke bestimmt waren:

- AX: Akkumulator. Wurde als Operandenregister für arithmetische Operationen benutzt.
- CX: Counter. Wird bei Schleifeninstruktionen als Zählregister verwendet.
- DX: Data. Wurde als Operandenregister für arithmetische Operationen benutzt.
- BX: Base. Diente als Zeigerregister für indirekte Speicherzugriffe.
- SP: Stack Pointer. Zeigt auf die aktuelle Adresse des nächsten freien Stack-Elements.
- BP: Base Pointer. Enthält die Adresse des Prozedurrahmens der aktuellen Funktion.
- SI: Source Index. Enthält bei String- und Speicher-Operationen die Quelladresse.
- DI: Destination Index. Enthält bei String- und Speicher-Operationen die Zieladresse.

Diese Register haben eine Breite von 16 Bit. Bei den ersten 4 Registern lassen sich das höherwertige Byte (High-Byte) und niederwertige Byte (Low-Byte) einzeln ansprechen, dazu wird das Suffix L beziehungsweise H statt X verwendet. Das Register AX besteht also aus den Teilen AH und AL.

Bei der Einführung neuerer Generationen fielen viele Beschränkungen der ersten 4 Register weg, sodass nun alle Register als Quelle oder Ziel arithmetischer Operationen dienen können. Die historischen Namen wurden allerdings aus Kompatibilitätsgründen beibehalten. Mit dem Intel 80386-Prozessor wurde die Breite der 8 Register auf 32 Bit erhöht. Aus Gründen der Kompatibilität haben die Register unter den obigen Bezeichnungen allerdings weiterhin nur eine Breite von 16 beziehungsweise 8 Bit; zur Verwendung der vollen 32 Bit wird dem Registernamen ein E (für extended) vorangestellt, beispielsweise EAX. Tabelle 2 zeigt eine Übersicht über die vorhandenen Register und deren unterschiedliche Namen.

Befehle werden durch sogenannte *Opcodes* codiert. Dabei handelt es sich um ein oder mehrere Bytes, die einen Befehl darstellen. Die textuelle Repräsentation eines Opcodes wird als

Assembler-Anweisung bezeichnet und hat in der Intel-Syntax folgenden Aufbau:
`<mnemonic> [op1][, op2][, op3]`

Als *Mnemonic* wird ein aus wenigen Buchstaben bestehender Name einer Anweisung bezeichnet, beispielsweise `cmp` für die Vergleichsanweisung. Bei Befehlen mit Quell- und Zieloperand ist der erste Operand in der Regel das Ziel. So kopiert die Anweisung `mov eax, ebx` den Inhalt von Register EBX in das Register EAX. Eine Dereferenzierung von Adressen im Speicher wird in eckigen Klammern angegeben. Davor wird zusätzlich angegeben, wie breit das an der Adresse dereferenzierte Datum ist, beispielsweise BYTE (8 Bit), WORD (16 Bit) oder DWORD (32 Bit). Der folgenden Befehl kopiert beispielsweise den an der Adresse 0x402000 gespeicherten 16-Bit-Wert ins Register AX (hexadezimale Zahlen werden mit dem Suffix h statt dem Präfix 0x angegeben): `mov ax, word ptr [402000h]`. Weitere Befehle sind im Anhang A erläutert.

Ein ausführbares Programm benutzt üblicherweise Funktionen des Betriebssystems oder aus anderen Bibliotheken, um mit seiner Umgebung zu interagieren. Beim Starten eines Programms muss das Betriebssystem diese Bibliotheken laden, damit das Programm sie benutzen kann. Dabei muss das Problem gelöst werden, dass der interne Aufbau der verwendeten Bibliotheken über verschiedene Versionen der Bibliothek nicht konstant ist – insbesondere ist also nicht bekannt, an welcher Adresse sich eine benötigte Funktion befinden wird. Das gleiche gilt für die Funktionen, die das Betriebssystem selbst bereitstellt. Ein anderes Problem ist, dass ein Programm seine eigene Adresse im physikalischen Speicher bei der Ausführung nicht kennt – es sind also bereits Aufrufe von Funktionen mit absoluten Adressen innerhalb des Programms problematisch.

Moderne Betriebssysteme lösen Probleme dieser Art durch die Verwendung virtuellen Speichers: Jedes Programm wird in einen eigenen, isolierten Adressbereich geladen, der als *virtueller Speicherbereich* bezeichnet wird, da die dortigen Adressen nicht den Adressen des real vorhandenen Hauptspeichers entsprechen. Diese Technik ist auf x86-Prozessoren seit dem 80286-Prozessor möglich – durch die Einführung der *Memory Management Unit* (MMU) kann das Betriebssystem für jeden Prozess getrennt konfigurieren, wie virtuelle Adressen auf den realen Hauptspeicher abgebildet werden sollen. Dadurch ist es möglich, dass Programme selbst angeben können, an welche virtuelle Adresse sie geladen werden sollen. Dies löst alle oben beschriebenen Probleme: Bibliotheken und Betriebssystemfunktionen werden an konstante Adressen des virtuellen Speichers geladen, sodass sie unabhängig von der Version oder der Adresse im Hauptspeicher immer an derselben virtuellen Adresse verfügbar sind. Unterschiedliche Instanzen desselben Programms benutzen denselben virtuellen Adressraum, der durch die MMU an unterschiedliche Adressen im realen Speicher abgebildet wird. Der virtuelle Speicher eines Programms ist dabei üblicherweise in unterschiedliche Sektionen unterteilt, die in Tabelle 3 beschrieben sind. Die dort mit `.data` bezeichnete Sektion wird im folgenden als *Datensegment* bezeichnet, die Sektion `.text` als *Codesegment*.

Ein Programm kann neben den beschriebenen Sektion zur Laufzeit weiteren Speicher vom Betriebssystem anfordern. Dieser Speicherbereich wird als *Heap* bezeichnet. Um Speicher im Heap zu reservieren, vergrößert das Programm zur Laufzeit sein Datensegment durch einen Betriebssystemaufruf. Bei einer Anforderung zur Vergrößerung des Datensegments prüft das Betriebssystem, ob noch genügend freier Hauptspeicher oder Auslagerungsspeicher (Swap)

Sektion	Verwendung	Rechte
.text	Programmcode	ausführbar, lesbar
.data	Initialisierte Daten: Zeichenketten, Konstanten usw.	lesbar, schreibbar
.bss	Uninitialisierte Daten: wird beim Laden mit 0 initialisiert und belegt keinen Platz im Programm	lesbar, schreibbar

Tabelle 3: Die Tabelle zeigt typische Sektionen in Binärformaten für Maschinencode. Sie werden verwendet, um unterschiedliche Rechte für den virtuellen Speicher zu verwenden, damit sich der Programmcode beispielsweise nicht zur Laufzeit selbst ändern kann. Ein weiterer Vorteil ist, dass eine Sektion im virtuellen Speicher angelegt werden kann, die in der eigentlichen Datei gar nicht abgebildet ist und vom Betriebssystem mit Nullen initialisiert wird. So bleiben die Dateien kleiner, da alle nicht-initialisierten Variablen in dieser Sektion untergebracht werden können.

Abbildung 4: Im virtuellen Adressbereich eines Programms der x86-Architektur wachsen Datensegment und Stack aufeinander zu: Das Datensegment wächst bei Nutzung des Heaps von kleineren zu größeren Adressen, während neue Elemente des Stacks auf kleiner werdende Adressen gelegt werden. Auf diese Weise können beide Speicherbereiche den freien Speicher dynamisch aufteilen.

vorhanden ist und konfiguriert die MMU des Prozessors so, dass die reale Adresse des neuen Speicherblocks den virtuellen Block des Datensegments fortsetzt.

Einen besonderen Speicherbereich bildet darüber hinaus der *Stack*. Dieser wird beim Erzeugen eines Prozesses oder Threads vom Betriebssystem reserviert. Auf dem Stack können mit den Maschinencode-Befehlen `push` und `pop` Werte mit einer Breite von 32 Bit abgelegt und von der Spitze gelesen werden. Hierzu legt das Betriebssystem für jeden erstellten Thread im Register ESP die *letzte* Adresse des virtuellen Speichers ab, die für den Stack genutzt werden darf. Die Anweisungen `push` und `pop` haben folgende Effekte:

push a dekrementiert (!) den Wert von ESP um 4 (d.h. um einen 32-Bit-Platz) und kopiert den Operanden a an die neue Adresse. In C-Syntax entspricht dies der Anweisung `*(--esp) = a`

pop a kopiert den Inhalt der Adresse in ESP nach *a* und inkrementiert (!) den Wert von ESP um 4. In C-Syntax entspricht dies der Anweisung `a = *(esp++)`.

Der Stack wächst also von größeren Adressen zu kleineren. Der Grund dafür ist, dass Datensegment und Stack des Programms auf diese Weise den gleichen virtuellen Adressbereich nutzen können. Sie wachsen dabei aufeinander zu (vgl. Abbildung 4): Das Datensegment von kleinen Adressen zu größeren und der Stack von größeren zu kleineren. Der freie Bereich

dazwischen kann so je nach Bedarf für die Vergrößerung des Datensegments oder den Stack genutzt werden.

Neben dem Register ESP gibt es noch das Register EBP für die Verwaltung des Stacks. Hier kann die Adresse des Prozedurrahmens einer Funktion gespeichert werden, sodass nicht nur auf die Spitze, sondern auf beliebige Elemente des Prozedurrahmens zugegriffen werden kann. Abbildung 5 erläutert den Zusammenhang zwischen den Registern EBP und ESP.

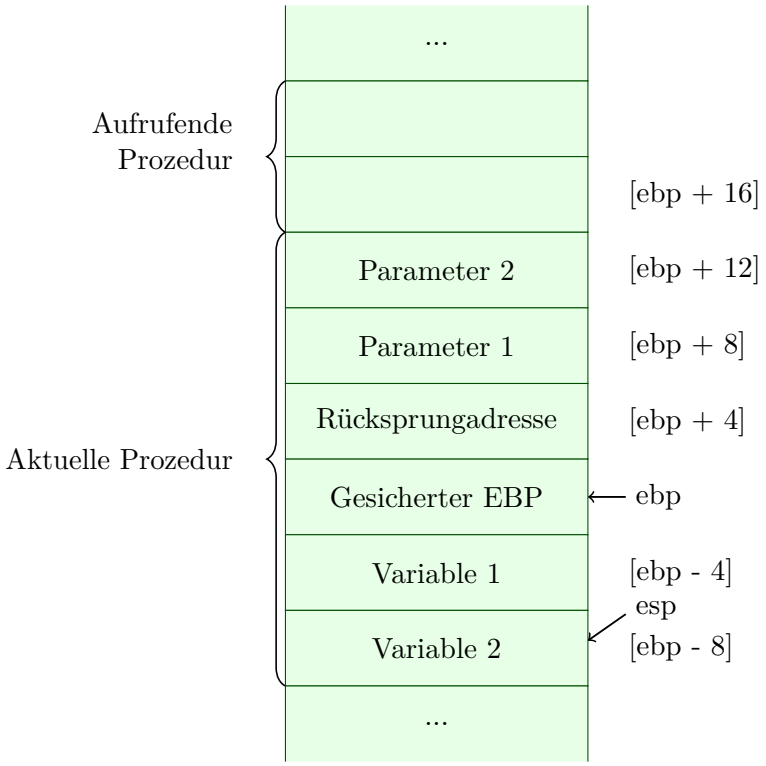


Abbildung 5: Die Abbildung zeigt den typischen Aufbau eines Prozedurrahmens der x86-Architektur unter Verwendung einer C-ähnlichen Sprache. Vor dem Aufruf einer Prozedur werden zuerst die Parameter auf den Stack gelegt. Die Reihenfolge ist dabei in C-Programmen rückwärts, weil Prozeduren in C eine variablen Anzahl von Parametern haben können (die ...-Notation). Durch die umgekehrte Reihenfolge wird sichergestellt, dass die Parameter vor der variablen Parameterliste eine eindeutige Position haben. Nach den Parametern wird durch die `call`-Anweisung die Rücksprungadresse auf den Stack gelegt. Danach richtet die aufgerufene Prozedur ihren Rahmen ein: Die Spitze des Stacks wird in das Register `ebp` kopiert, damit innerhalb des Rahmens relativ adressiert werden kann. Parameter 1 ist dadurch immer über `[ebp + 8]` ansprechbar, ebenso alle anderen Parameter und Variablen. Die Spitze des Stacks ist im Register `esp`, das der Prozessor bei jeder Stack-Operation anpasst. Durch eine Subtraktion (!) von `esp` kann somit Platz für lokale Variablen reserviert werden. Parameter für weitere Prozeduraufrufe werden dadurch an Adressen unterhalb der lokalen Variablen angelegt. Beim Verlassen der Prozedur wird `esp` auf den Wert von `ebp` zurückgesetzt, wodurch die Stack-Spitze wieder auf den alten Wert zeigt. Nun wird der gesicherte Wert von `ebp` wiederhergestellt, damit die aufrufende Prozedur ihren Rahmen wieder relativ adressieren kann. Der `ret`-Befehl kehrt schließlich aus der Prozedur zurück, indem an die Rücksprungadresse gesprungen und deren Wert vom Stack entfernt wird.

2 Reverse Engineering: Methoden des Angreifers

Unter *Reverse Engineering* werden Methoden verstanden, die Details der Implementierung eines Programms ohne Vorhandensein des Hochsprachenquellcodes gewinnen. Beim Vorgehen wird zwischen zwei grundlegenden Methoden unterschieden: In der Black-Box-Analyse wird ein Programm oberflächlich durch Ausführung untersucht, während in der White-Box-Analyse der Maschinencode analysiert wird. Rückschlüsse aus einer White-Box-Analyse werden häufig durch eine erneute Black-Box-Analyse überprüft, sodass der Gesamtvorgang üblicherweise iterativ abläuft.

2.1 Black-Box-Analyse

Bei einer Black-Box-Analyse wird das zu analysierende Programm ausgeführt und während der Ausführung beobachtet. Der Name dieser Analysemethode leitet sich aus der Sichtweise her, dass das Programm als *Black Box* mit unbekannter Funktionsweise betrachtet wird, die aus Eingaben Ausgaben berechnet. Im Rahmen einer solchen Analyse können folgende, für die weitere Analyse wichtige Eigenschaften eines Programms ermittelt werden:

- Ausgaben: die Ausgaben eines Programms liefern wichtige Einstiegspunkte für eine folgende White-Box-Analyse. Wenn das zu untersuchende Programm beispielsweise unmittelbar vor der Berechnung eines Ergebnisses die Meldung „Bitte warten, Ergebnis wird berechnet“ anzeigt, so ist es wahrscheinlich, dass der Code zur Berechnung in der Nähe des Codes zur Ausgabe der Meldung aufgerufen wird. Bei der Umgehung eines Kopierschutzes könnte eine Fehlermeldung wie „Seriennummer ungültig“ oder „Bitte Original-DVD einlegen“ dem Angreifer helfen, die Stelle im Maschinencode zu finden, die nach der erfolglosen Prüfung des Kopierschutzes ausgeführt wird.
- Ausführungszeit: Beobachtungen zum zeitlichen Verhalten des Programms können Rückschlüsse auf die Implementierung erlauben. Es könnte beispielsweise einige Eingaben für das Programm geben, bei denen die Reaktionszeit des Programms von den durchschnittlichen Reaktionszeiten anderer Eingaben abweicht. Solche Eingaben können in einer anschließenden White-Box-Analyse bei der Analyse helfen, da sie möglicherweise zu Ausführungspfaden im Maschinencode führen, die durch andere Eingaben nicht erreicht werden.

Formal wird in der Black-Box-Analyse das Programm als Funktion P aufgefasst, die eine Eingabe x auf den Funktionswert $P(x)$ abbildet. Die Eingabe enthält dabei die gesamte Umgebung des Programms, also Eingaben des Benutzers, Ergebnisse von Betriebssystemaufrufe und so weiter. Während der formalen Black-Box-Analyse können die Funktionswerte von P für gegebene x beobachtet werden.

Die Black-Box-Analyse ist in der Regel nur der erste Schritt, um grundlegende Erkenntnisse über das Verhalten des Programms zu gewinnen. Allein aus der Beobachtung lässt sich ein Programm üblicherweise nicht vollständig rekonstruieren. Auch eine Manipulation des Programms ist auf diese Weise nicht möglich.

2.2 White-Box-Analyse

Eine White-Box-Analyse wird durchgeführt, um das in der Black-Box-Analyse beobachtete Verhalten des Programms so weit nachvollziehen zu können, dass eine Rekonstruktion oder Manipulation möglich wird. Diese Form der Analyse geschieht auf Basis des Maschinencodes des Programms. Formal liegt für diese Analyse eine codierte Turingmaschine vor, die die Funktion P des Programms implementiert.

Für die White-Box-Analyse gibt es zwei grundlegende Ansätze. Die Turingmaschine für P kann einerseits für gegebene x simuliert werden, sodass die Berechnung $P(x)$ beobachtet werden kann (dynamische Analyse). Andererseits kann die Turingmaschine selbst analysiert werden, um daraus Erkenntnisse über die Berechnung zu erhalten.

2.2.1 Statische Analyse

Eine statische Analyse wird auf dem Maschinencode des Programms durchgeführt. Dieser wird durch einen *Disassembler* erhalten, der ein ausführbares Maschinenprogramm in ein Assembler-Listing übersetzt. Obwohl das Format von ausführbaren Programmdateien bekannt ist und auch der Befehlssatz des Prozessors dokumentiert ist, kann ein Disassembler im Allgemeinen den Maschinencode nicht in der Form zurückgewinnen, die durch einen Assembler in dasselbe Programm übersetzbar wäre. Dies hat unter anderem folgende Gründe:

- Datentypen: auf Ebene der Maschine wird beim Inhalt von Speicher und Registern nicht zwischen verschiedenen Datentypen unterschieden. Wenn beim Disassemblieren der Befehl `push 402345h` decodiert wird, so ist es für den Disassembler im Allgemeinen nicht berechenbar, ob es sich bei der Konstante um einen Integer-Wert oder um einen Zeiger auf eine Speicherstelle handelt. Falls die Zahl eine gültige Adresse im virtuellen Adressraum der Anwendung ist, so könnte mit einer Heuristik geprüft werden, ob die Nutzung dieser Adresse an der Stelle im Code möglich wäre. Aber selbst, falls davon mit Sicherheit auszugehen ist – am Ziel der Adresse wird sich wiederum eine Zahl befinden, bei der der Disassembler erneut prüfen muss, ob es sich dabei um eine Adresse handelt. In der Programmiersprache C ist dies gut nachzuvollziehen, da auch hier Zahlen beliebig interpretiert und über Typecasts umgewandelt werden können. Die C-Typen `char *` und `int` sind in der x86-Architektur beispielsweise identisch, da ein Zeiger genau die Größe von int-Zahlen besitzt. Für den Disassembler ist die Kenntnis des Datentypes aber wichtig, um eine Verbindung zwischen Zeichenketten im Datensegment und den verwendenden Instruktionen im Codesegment herzustellen. Diese Verbindung ist für den Angreifer wichtig, um von einer in der Black-Box-Analyse gefundenen Zeichenkette schnell zu den Stellen im Maschinencode zu gelangen, die auf diese Zeichenkette zugreifen.
- Indirekte Sprünge: Eine Anweisung wie `call eax` ist problematisch für den Disassembler, da der Wert des Registers EAX möglicherweise nicht aus dem Umfeld der Anweisung hervorgeht. Wenn der Wert beispielsweise aus einer dynamischen Datenstruktur entsteht, deren Werte erst zur Laufzeit berechnet werden, so ist es für den

Disassembler ohne Simulation großer Teile des Programms nicht möglich, das Ziel der `call`-Anweisung zu bestimmen. Falls die auf diese Weise aufgerufene Prozedur *ausschließlich* auf indirektem Weg aufgerufen wird, so bleibt sie dem Disassembler verborgen und kann nur durch Heuristiken gefunden werden, die beispielsweise nach dem Disassemblieren alle nicht besuchten Stellem im Codesegment der Anwendung untersuchen. Anweisungen mit indirekten Sprüngen und Prozeduraufrufen entstehen durch Hochsprachenkonstrukte wie Funktionszeiger, switch-case-Anweisungen und überschriebene Methoden bei objektorientierten Programmen.

Es gibt daher große Qualitätsunterschiede zwischen verschiedenen Disassemblern. Einfache Disassembler wie `objdump` (Teil der GNU `binutils`) verfügen über keinerlei Heuristiken und sind daher oft nicht für eine ausführliche White-Box-Analyse geeignet. Als Marktführer unter kommerziellen Disassemblern gilt *IDA Pro* von Hex-Rays [HR]. Dabei handelt es sich um einen kommerziellen Disassembler, der über viele konfigurierbare Heuristiken, Visualisierungen und eine Python-Schnittstelle zur automatisierten Analyse verfügt.

Da es keinen freien Disassembler mit ähnlichem Funktionsumfang gibt, wurde im Rahmen eines Masterprojekts der Disassembler *Kianxali* [Wil14] entwickelt, der über den für diese Arbeit nötigen Funktionsumfang verfügt und über die Skriptsprache Ruby zur automatisierten Analyse eingesetzt werden kann. Mit Kianxali können Maschinenprogramme für die Betriebssysteme Windows, Linux und OS X für die Prozessorarchitekturen x86 und x86-64 analysiert werden.

2.2.2 Dynamische Analyse

Bei der dynamischen Analyse wird das zu analysierende Programm innerhalb eines *Debuggers* ausgeführt. Dieser führt das Programm in einer Umgebung aus, die eine Beobachtung des internen Zustands des Programms zur Laufzeit ermöglicht. Der Benutzer des Debuggers kann das Programm zu beliebigen Zeitpunkten pausieren und dabei den Laufzeit-Speicher des Programms untersuchen. Ein Debugger enthält üblicherweise einen Disassembler, um den Maschinencode der lokalen Umgebung der Pausierung anzeigen zu können. Weiterhin ist es möglich, das Programm durch *Break Points* an beliebigen Stellen automatisch zu pausieren, um dann im Debugger die Inhalte der Register des Prozessors anzusehen oder zu manipulieren. Auf diese Weise kann durch den Angreifer schnell überprüft werden, welche Auswirkung eine erzwungene Änderung des Kontrollflusses hat. Die dynamische Analyse hat allerdings den Nachteil, dass nur Code analysiert werden kann, der auch tatsächlich zur Ausführung gelangt. Eine partielle oder vollständige Simulation des Programms durch einen Interpreter zählt ebenfalls zur dynamischen Analyse. Insbesondere bei Schadcode oder Programmen, die auf zeitliche oder netzwerkbasierte Ereignisse reagieren, ist es nur schwierig möglich, eine bestimmte Stelle im Code des Programms auch tatsächlich zur Ausführung zu bringen. Die dynamische Analyse wird daher in der Regel ergänzend zur statischen Analyse durchgeführt.

2.3 Erhaltene Informationen

Durch die oben beschriebenen Analysemethoden kann der Angreifer bereits genug Informationen erhalten, um ein Programm anzugreifen.

2.3.1 Kontrollflussgraph

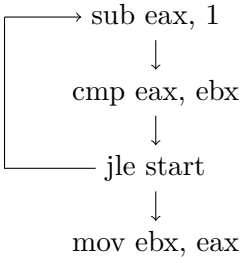
Ein für Angreifer besonders interessantes Detail ist der Kontrollflussgraph eines Programms. Darunter wird in dieser Arbeit der gerichtete Graph $G_{cfg} = (V, E)$ verstanden, bei dem jede Maschinencodeanweisung des Programms ein Knoten ist. Eine Kante zwischen v_i und v_j wird gesetzt, wenn der Prozessor nach der Ausführung des Befehls v_i als nächstes den Befehl v_j ausführen könnte. Er enthält also alle Anweisungen des Maschinenprogramms. Der Graph kann dabei aus mehreren unabhängigen Zusammenhangskomponenten bestehen. Der Grund dafür ist, dass einige Funktionen im Code indirekt aufgerufen werden, indem beispielsweise die Adresse einer Funktion in ein Register geladen und dessen Inhalt als Ziel einer `call`-Anweisung genutzt wird. Ein solcher Aufruf wird vom Kontrollflussgraph nicht erfasst, da aufgrund der fehlenden Datentypen nie sicher berechnet werden kann, ob eine Konstante in einem Register tatsächlich eine Zieladresse für einen Sprung ist oder nur als Konstante für eine Berechnung dient.

Der Kontrollflussgraph wird durch den Disassembler erstellt – dabei ist allerdings zu beachten, dass wegen der oben beschriebenen Gründe Kanten und Knoten fehlen können. Abbildung 6 zeigt einen beispielhaften Kontrollflussgraph.

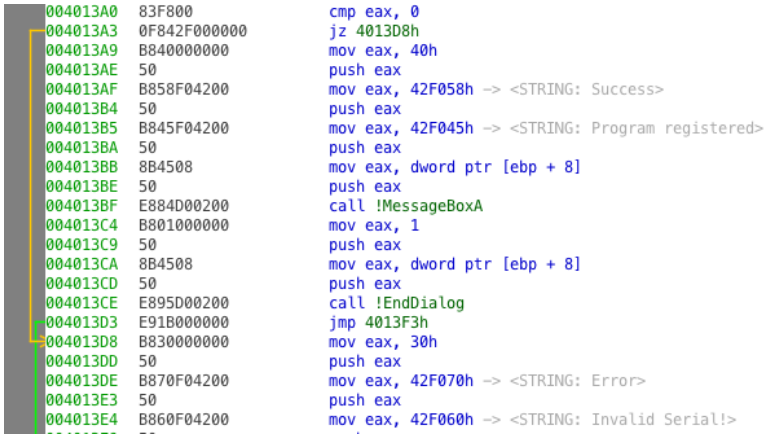
Der Kontrollflussgraph ist wichtig für den Angreifer, da er über diesen rückwärts durch den Assembler-Code navigieren kann. In der Regel navigiert er vom Symptom (z.B. einer Fehlermeldung) zur Ursache (z.B. der Prüfung einer Seriennummer). Grafische Disassembler visualisieren den Kontrollfluss daher oft über Pfeile.

2.3.2 Informationen über Nutzung externer Bibliotheken

Ein Maschinenprogramm läuft nie autark – zur Kommunikation mit der Umgebung werden Funktionen des Betriebssystems benötigt. Selbst das Beenden des Prozesses benötigt schon einen Aufruf an das Betriebssystem, um die Freigabe des Prozesses anzufordern. Neben der API des Betriebssystems kann das Programm aber auch auf externe Programmbibliotheken zugreifen, die Maschinencode bereitstellen. Unter Windows wird die API des Betriebssystems ebenfalls als Programmbibliothek bereitgestellt, sodass der genutzte Mechanismus identisch ist. Programmbibliotheken und API werden unter Windows über *Dynamic Link Libraries* in Form von DLL-Dateien bereitgestellt. Diese enthalten neben dem Maschinencode zur Umsetzung der bereitgestellten Funktionen auch eine Tabelle mit den Namen der bereitgestellten Funktionen, um diese für eine Verwendung zu exportieren. So gibt es in der Windows-API beispielsweise die Bibliothek `user32.dll`, in der Funktionen zur grafischen Interaktion mit dem Benutzer bereitgestellt werden. Darin gibt es unter anderem die Funktion `MessageBoxA` zur Anzeige einer ASCII-Zeichenkette in Form eines Dialogs.



(a) Kontrollflussgraph



(b) Darstellung eines Kontrollflussgraphen in Kianxali

Abbildung 6: Aus einem Kontrollflussgraphen ist für den Angreifer schnell ersichtlich, dass es sich bei dem Ausschnitt (a) um eine Schleife handelt, da es einen Zyklus im Graph gibt, der zur Wiederholung von Code führt. Die Darstellung eines Kontrollflusgraphen in Kianxali verzichtet auf Kanten zwischen benachbarten Knoten, sodass nur Verzweigungen dargestellt werden. Hier ist gezeigt, wie der Angreifer den Graphen rückwärts analysiert, um von der Fehlermeldung an Adresse 4013DE über den orangen Pfeil zur Verzweigung an Adresse 4013A3 gelangt, die über Erfolg oder Fehler entscheidet.

Wenn ein Programm diese Funktionen verwenden soll, so fügt der Compiler eine Tabelle zum Importieren der fremden Funktionen in das Maschinenprogramm ein. Mit dieser werden die benötigten Funktionen über den Namen der DLL-Datei sowie den Namen der benötigten Funktion beim Laden des Programms vom Betriebssystem bereitgestellt. Diese Informationen kann ein Disassembler nutzen, um alle externen Funktionsaufrufe mit den korrekten Namen der Funktion zu annotieren, sodass anstelle von Anweisungen wie `call 501000h` eine annotierte Variante wie `call MessageBoxA` angezeigt werden kann. Für den Angreifer sind diese Annotationen wertvoll, da er anhand der API-Aufrufe innerhalb einer Funktion leichter nachvollziehen kann, wozu die analysierte Funktion benutzt werden könnte. Der Disassembler Kianxali unterstützt diese Annotationen (siehe z.B. im Code von Abbildung 6). Es können mithilfe des Kontrollflussgraphs ebenfalls alle Verwendungen einer externen Funktion gefunden werden, sodass beispielsweise alle Stellen im Maschinencode gefunden werden können, die bestimmte API-Funktionen des Betriebssystems nutzen.

2.3.3 Rekonstruktion einzelner Funktionen

Einzelne Funktionen können vom Angreifer manuell in eine äquivalente Hochsprachenfunktion überführt werden. Dies ist häufig gewünscht, um einen Algorithmus in einer eigenen Anwendung zu verwenden, beispielsweise im Rahmen von Industriespionage. Die Programmiersprache C ist dafür am besten geeignet, da sich fast alle Instruktionen des x86-Befehlssatzes als eine oder mehrere C-Anweisungen umsetzen lassen. Insbesondere die Möglichkeit zur

Konvertierung von Integern zu Zeigern erweist sich dabei oft als nützlich. Auch das Vorhandensein einer `goto`-Anweisung ist für die Rekonstruktion von Assembler-Code sinnvoll, damit Sprunganweisung des Maschinencodes direkt übernommen werden können. Die Umsetzung der Rekonstruktion geschieht in der Regel manuell, weil der Angreifer durch die Umsetzung meist auch eine tiefere Einsicht in den Effekt der Funktion erhalten möchte. Der Disassembler IDA Pro enthält einen Decompiler, der bei diesem Vorhaben unterstützen kann. Eine vollständig automatisierte Dekompilierung ist allerdings nicht möglich, die Gründe dazu sind dieselben, die wie oben beschrieben bereits einen Disassembler unpräzise machen. Ein manuell durchgeführten Algorithmus zur Rekonstruktion einer Funktion aus ihrem Maschinencode könnte folgendermaßen ablaufen:

1. deklariere die lokalen Variablen `eax, ebx, ecx, edx, esi` und `edi` als `unsigned int`, um die Verwendung von Registern nachzubilden
2. analysiere die Parameter der Funktion anhand von Zugriffe auf positive Offsets auf `EBP` (vgl. Abbildung 5 auf Seite 13) und setze sie als Parameter der C-Funktion um
3. führe für jeden im Prozedurrahmen genutzten Speicherplatz (negatives Offset zu `EBP`, vgl. Abbildung 5) eine Variable ein: `ebp4, ebp8, ebp12, ...`
4. setze Sprunganweisungen als `goto`-Anweisung um und führe dazu Label für jedes Sprungziel ein
5. setze Assembler-Konstrukte durch äquivalente C-Konstrukte um
6. optional: Vereinfache den erhaltenen Code durch Ersetzung von `goto`-Anweisungen durch Schleifen und Blöcke

2.4 Angriff der Beispielanwendung

Um das Zusammenspiel der Analysemethoden zu verdeutlichen, soll der Kopierschutz der Beispielanwendung angegriffen werden.

2.4.1 Black-Box-Analyse

Dazu wird sie zunächst einer Black-Box-Analyse unterzogen, indem sie ausgeführt wird und wichtige Meldungen notiert werden:

- wenn der eingegebene Lua-Quellcode länger als der beim Start angezeigte Beispielcode ist, wird die Meldung „This demo version only supports short programs“ angezeigt (siehe Abbilung 7). Das Aussehen des dazu benutzten Dialogs lässt vermuten, dass dazu die Funktion `MessageBoxA` der Windows-API verwendet wurde.
- der Dialog zur Registrierung enthält ein Feld für den Namen des Lizenznehmers und eine zugehörige Seriennummer. Wird der Dialog mit dem fiktiven Lizenznehmer „Angreifer“ und der Seriennummer 1234 gefüllt, erscheint nach Betätigung des OK-Buttons die Fehlermeldung „Invalid Serial!“ (siehe Abbildung 8).

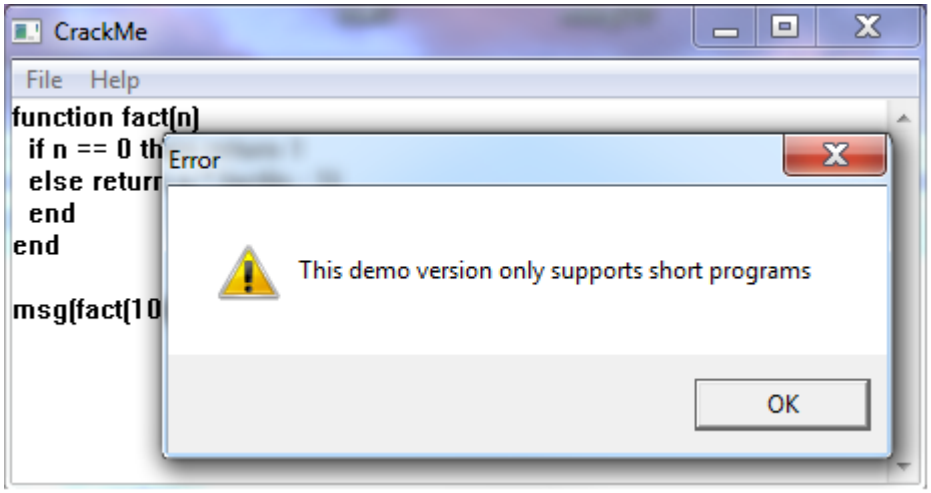


Abbildung 7: Wenn die Länge des Lua-Quellcodes eine bestimmte Länge überschreitet, zeigt die Beispielanwendung eine Fehlermeldung. Diese wurde mithilfe einer Black-Box-Analyse ermittelt, bei der das Programm ohne Änderungen oder andere Eingriffe gestartet und benutzt wurde.

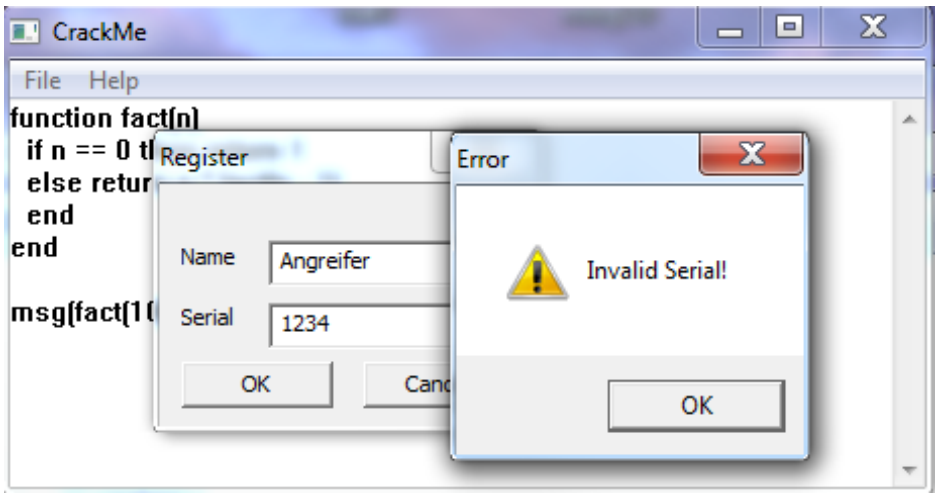


Abbildung 8: Der Kopierschutz der Beispielanwendung zeigt einen Dialog, in dem der Benutzer seinen Namen und eine zugehörige Seriennummer eingegeben kann. In der Black-Box-Analyse wurde ein fiktiver Name und eine ausgedachte Seriennummer eingegeben, bei der das Programm eine Fehlermeldung zeigt.

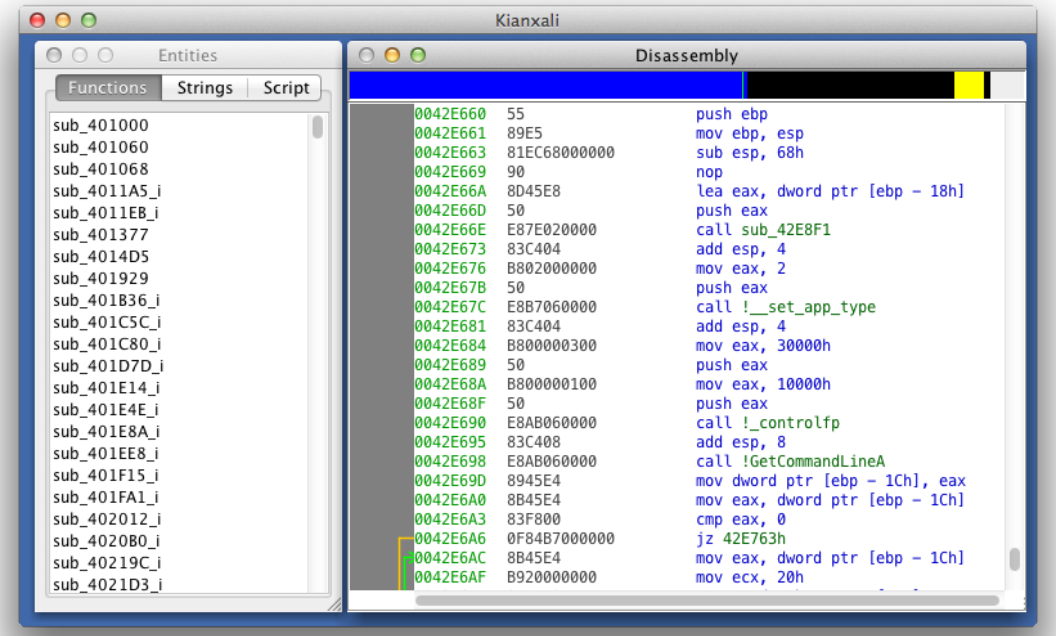


Abbildung 9: Die Abbildung zeigt die Beispielanwendung im Disassembler Kianxali. Die Anzeige des Maschinencodes (rechts) erfolgt hier dreispaltig: Zuerst die virtuellen Speicheradressen der Anweisungen, dann die hexadezimale Darstellung des Opcodes der Anweisung und schließlich die disassemblierte Form als lesbarer Assembler-Code. Links befindet sich eine Liste aller disassemblierten Funktionen.

2.4.2 Statische Analyse

Um den Kopierschutz der Beispielanwendung anzugreifen und dadurch eine Freischaltung des Programms ohne gültige Seriennummer zu erreichen, erfolgt nun eine White-Box-Analyse in Form einer statischen Analyse mit dem Disassembler *Kianxali* [Wil14]. In diesem wird nach dem Laden der Anwendung der Maschinencode der Anwendung in Form eines Assembler-Listings dargestellt. Der Kontrollfluss des Programms lässt sich anhand von Pfeilen nachvollziehen, die bei Sprüngen auf das Sprungziel zeigen. Nach dem Disassemblieren zeigt Kianxali den Maschinencode des Startup-Codes des Programms (siehe Abbildung 9). Dessen Adresse ist im Header der Datei angegeben, hier beginnt die Ausführung des Programms.

Für einen Angriff des Kopierschutzes muss dessen Implementierung nun im Maschinencode gefunden werden. Da Kianxali über eine Heuristik zur Erkennung vom Zeichenketten in Programmen verfügt, kann diese Funktion genutzt werden, um über die im Rahmen der Black-Box-Analyse erhaltene Fehlermeldung „Invalid Serial!“ den Maschinencode zu finden, der diese Zeichenkette verwendet (siehe Abbildung 10). Die Umgebung dieses Maschinencodes wird von Kianxali folgendermaßen dargestellt:

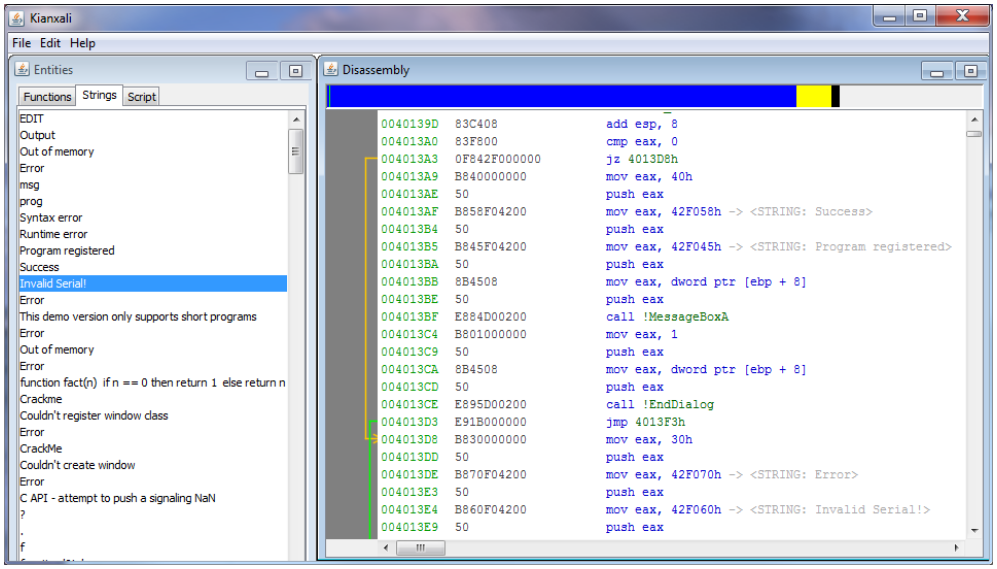


Abbildung 10: Die Abbildung zeigt einen Ausschnitt des Maschinencodes der Beispielanwendung. Durch die Heuristik des Disassemblers konnte die Verwendung der durch die Black-Box-Analyse erhaltene Fehlermeldung, die bei der Eingabe einer ungültigen Seriennummer angezeigt wird (vgl. Abbildung 8), im Maschinencode gefunden werden.

```
1  00401398 call sub401052
2  0040139D add esp, 8
3  004013A0 cmp eax, 0
4  004013A3 jz 4013D8h
5  004013A9 mov eax, 40h
6  004013AE push eax
7  004013AF mov eax, 42F058h ; -> <STRING: Success>
8  004013B4 push eax
9  004013B5 mov eax, 42F045h ; -> <STRING: Program registered>
10 004013BA push eax
11 004013BB mov eax, dword ptr [ebp + 8]
12 004013BE push eax
13 004013BF call !MessageBoxA
14 004013C4 mov eax, 1
15 004013C9 push eax
16 004013CA mov eax, dword ptr [ebp + 8]
17 004013CD push eax
18 004013CE call !EndDialog
19 004013D3 jmp 4013F3h
20 004013D8 mov eax, 30h
21 004013DD push eax
22 004013DE mov eax, 42F070h ; -> <STRING: Error>
23 004013E3 push eax
24 004013E4 mov eax, 42F060h ; -> <STRING: Invalid Serial!>
25 004013E9 push eax
26 004013EA mov eax, dword ptr [ebp + 8]
27 004013ED push eax
28 004013EE call !MessageBoxA
```

In Zeile 24 wird die Adresse der Meldung „Invalid Serial!“ ins Register `EAX` geladen und in der folgenden Zeile auf den Stack gelegt. Insgesamt handelt es sich bei den Zeilen 20 bis 28 um den folgenden Funktionsaufruf: `MessageBoxA(ebp8, "Invalid Serial", "Error", 0x30)`. Bei `ebp8` handelt es sich um eine lokale Variable der aufrufenden Funktion. Aus der Dokumentation der Windows-API [Mic] ist ersichtlich, dass die Funktion `MessageBoxA` einen Dialog mit einem Text und einem Titel auf dem Bildschirm anzeigt. Der erste Parameter enthält eine Identifikationsnummer des Fensters, das durch den Dialog blockiert werden soll. Der letzte Parameter legt das Aussehen des Dialogs fest, 0x30 entspricht dabei dem Wert der Konstante `MB_ICONEXCLAMATION`, die den Dialog mit einem grafischen Ausrufezeichen versieht. Dies entspricht genau dem in Abbildung 8 gezeigten Verhalten. Dieser Funktionsaufruf wird durch einen bedingten Sprung in Zeile 4 ausgelöst. Die Bedingung des Sprungs ergibt sich durch den Code in Zeile 3 als Vergleich des Registers `EAX` mit dem Wert 0. Das Register `EAX` wird üblicherweise für Rückgabewerte von Funktionsaufrufen verwendet, sodass der Sprung vom Ergebnis des Funktionsaufrufs in Zeile 1 abhängt: Gibt die Funktion 0 zurück, so wird die Fehlermeldung angezeigt. Anderenfalls wird der Code ab Zeile 5 ausgeführt, der offenbar eine Erfolgsmeldung anzeigt. Es ist also anzunehmen, dass die in Zeile 1 aufgerufene Funktion eine eingegebene Seriennummer überprüft und im Falle eines Fehlers den Wert 0 zurückgibt. Der Maschinencode dieser Funktion wurde ebenfalls disassembliert und wird schrittweise erläutert:

```
1  00401052 push ebp
2  00401053 mov ebp, esp
3  00401055 sub esp, 10h
4  0040105B nop
```

Hierbei handelt es sich um einen Funktionsprolog, bei dem 16 Bytes im Prozedurrahmen reserviert werden.

```
1  0040105C mov eax, dword ptr [ebp + 8]
2  0040105F push eax
3  00401060 call !strlen
4  00401065 add esp, 4
5  00401068 mov dword ptr [ebp - 4], eax
```

Dieser Abschnitt kopiert die Adresse des ersten Funktionsarguments in das Register `EAX`, übergibt sie auf dem Stack der Funktion `strlen` und kopiert die Rückgabe in die lokale Variable `ebp4`. Dies kann in C folgendermaßen realisiert werden:

```
int ebp4 = strlen(arg1);
```

```
1  0040106B mov eax, 0
2  00401070 mov dword ptr [ebp - 0Ch], eax
```

Die lokale Variable `ebp12` erhält den Wert 0:

```
int ebp12 = 0;
```

```
1  00401073 mov eax, dword ptr [ebp - 4]
2  00401076 cmp eax, 0
3  00401079 jnz 401089h
4  0040107F mov eax, 0
5  00401084 jmp 401113h
```

Der Wert der Variable `ebp4` wird mit 0 verglichen. Abhängig davon wird unterschiedlich verzweigt. Da die Adresse 401089h unmittelbar hinter Zeile 5 folgt, kann dies in C folgendermaßen beschrieben werden:
`if(ebp4 == 0) {eax = 0; goto l_401113;}`

```
1  00401089 mov eax, 10h
2  0040108E push eax
3  0040108F mov eax, 0
4  00401094 push eax
5  00401095 mov eax, dword ptr [ebp + 0Ch]
6  00401098 push eax
7  00401099 call !strtoul
8  0040109E add esp, 0Ch
9  004010A1 mov dword ptr [ebp - 10h], eax
```

Hier wird das zweite Argument der Funktion mittels der Funktion `strtoul` von einer hexadezimalen ASCII-Zeichenkette in eine entsprechende Zahl gewandelt, die in der Variable `ebp16` gespeichert wird:
`unsigned int ebp16 = strtoul(arg2, 0, 16);`

```
1  004010A4 mov eax, 0
2  004010A9 mov dword ptr [ebp - 8], eax
```

Die lokale Variable `ebp8` wird auf den Wert 0 gesetzt:
`int ebp8 = 0;`

```
1  004010AC mov eax, dword ptr [ebp - 8]
2  004010AF mov ecx, dword ptr [ebp - 4]
3  004010B2 cmp eax, ecx
4  004010B4 jnl 4010F2h
5  004010BA jmp 4010CAh
```

Hier werden die Variablen `ebp8` und `ebp4` miteinander verglichen, abhängig davon wird gesprungen:
`l_4010AC: if(ebp8 >= ebp4) goto l_4010F2; else goto l_4010CA;`

```
1  004010BF mov eax, dword ptr [ebp - 8]
2  004010C2 mov ecx, eax
3  004010C4 inc eax
4  004010C5 mov dword ptr [ebp - 8], eax
5  004010C8 jmp 4010ACh
```

Da die Adresse in Zeile 1 nicht vom Programmfluss des vorigen Abschnitts erreicht werden kann, wird sie vermutlich von einem der folgenden Abschnitte angesprungen, sodass direkt ein Label vorgesehen wird. Die Variable `ebp8` wird inkrementiert und der alte Wert im Register `ECX` behalten, dies kann in C via post-inkrement umgesetzt werden:
`l_4010BF: ecx = ebp8++; goto l_4010AC;`

```
1  004010CA mov eax, dword ptr [ebp + 8]
2  004010CD mov ecx, dword ptr [ebp - 8]
3  004010D0 add eax, ecx
```

Hier wird eine Zeigeroperation durchgeführt, die auf die Adresse des ersten Parameters den Wert der Variable `ebp8` addiert und das Ergebnis im Register `EAX` speichert:
`l_4010CA: eax = ((unsigned int) arg1) + ebp8;`

```
1  004010D2 mov ecx, dword ptr [ebp - 8]
2  004010D5 add ecx, 90h
3  004010DB movsx edx, byte ptr [eax]
4  004010DE xor edx, ecx
```

Hier wird `EAX` als Zeiger auf ein Byte-Array interpretiert und dessen Ziel mit `ebp8 + 0x90` exklusiv oder verknüpft, wobei das Ergebnis im Register `EDX` gespeichert wird:
`edx = *((unsigned char *) eax) ^ (ebp8 + 0x90);`

```
1  004010E0 mov eax, 0ABADF00Dh
2  004010E5 imul edx, eax
3  004010E8 mov eax, dword ptr [ebp - 0Ch]
4  004010EB xor eax, edx
5  004010ED mov dword ptr [ebp - 0Ch], eax
6  004010F0 jmp 4010BFh
```

Es erfolgt eine Multiplikation des Zwischenergebnis mit einer Konstante, bevor mit dem Wert von `ebp12` exklusiv oder verknüpft wird:
`edx *= 0xABADF00D; ebp12 ^= edx; goto l_4010BF;`

```
1  004010F2 mov eax, dword ptr [ebp - 0Ch]
2  004010F5 xor eax, 0CAFEh
3  004010FB mov dword ptr [ebp - 0Ch], eax
```

Es erfolgt eine Verknüpfung der Variable `ebp12` mit einer Konstante:
`l_4010F2: ebp12 ^= 0xCAFE;`

```
1  004010FE mov eax, dword ptr [ebp - 10h]
2  00401101 mov ecx, dword ptr [ebp - 0Ch]
3  00401104 cmp eax, ecx
4  00401106 mov eax, 0
5  0040110B setz al
6  0040110E jmp 401113h
7  00401113 leave
8  00401114 retn
```

Die Inhalte von `ebp16` und `ebp12` werden verglichen und das Ergebnis als Rückgabewert benutzt: `eax = (ebp12 == ebp16); l_401113: return eax;`

Nun kann aus diesen Fragmenten eine Funktion in einer Hochsprache geschrieben werden, die semantisch äquivalent zum analysierten Code ist. Der Angreifer kann den Hochsprachencode sukzessive vereinfachen, die verwendeten Variablen nach Aufdecken ihres Zwecks mit sinnvollen Bezeichnern versehen und auf diese Weise immer mehr Verständnis über die Funktionsweise der Prüfung der Seriennummer gewinnen.

Diese Vorgehensweise kann auch bei längeren Maschinencode-Fragmenten angewendet werden, selbst wenn dort andere Funktionen aufgerufen werden – diese können auf dieselbe Weise umsetzt werden, bis alle benötigten Teile des Programms rekonstruiert sind.

Es ergibt sich nach dieser Analyse vorerst folgende C-Funktion:

```
1  int sub401052(char *arg1, char *arg2) {
2      unsigned int eax, ecx, edx, ebp16;
3      int ebp4, ebp8, ebp12;
4
5      ebp4 = strlen(arg1);
6      ebp12 = 0;
7      if(ebp4 == 0) {
8          eax = 0;
9          goto l_401113;
10     }
11     ebp16 = strtoul(arg2, 0, 16);
12     ebp8 = 0;
13 l_4010AC:
14     if(ebp8 >= ebp4) {
15         goto l_4010F2;
16     } else {
17         goto l_4010CA;
18     }
19 l_4010BF:
20     ecx = ebp8++;
21     goto l_4010AC;
22 l_4010CA:
23     eax = ((unsigned int) arg1) + ebp8;
24     edx = *((unsigned char *) eax) ^ (ebp8 + 0x90);
25     edx *= 0xABADF00D;
26     ebp12 ^= edx;
27     goto l_4010BF;
28 l_4010F2:
29     ebp12 ^= 0xCAFE;
30     eax = (ebp12 == ebp16);
31 l_401113:
32     return eax;
33 }
```

Da der zweite Parameter der Funktion `strtoul` übergeben wird, die die übergebene Zeichenkette mit wählbarer Basis in einen Integer-Wert konvertiert, handelt es sich bei diesem Parameter vermutlich um die Seriennummer, die als hexadezimale Zeichenkette interpretiert wird, und beim ersten Parameter um den eingegebenen Namen. Die analysierte Funktion gibt 0 zurück, wenn der Name die Länge 0 hat (Zeile 7) oder ein Vergleich zwischen der Berechnung von `ebp12` und der eingegebenen Seriennummer (als Zahl in `ebp16` gespeichert) fehlschlägt. Die Informationen werden genutzt, um die Funktion leserlicher zu formulieren. Dabei werden Blöcke und zugehörige `goto`-Anweisungen miteinander verbunden, soweit dies direkt möglich ist:

```
1  int checkSerial(char *nameStr, char *serialStr) {
2      unsigned int eax, ecx, edx, serialNum;
3      int nameLen, ebp8, ebp12;
4      nameLen = strlen(nameStr);
5      ebp12 = 0;
6      if(nameLen == 0) {
7          return 0;
8      }
```

```
 9      serialNum = strtoul(serialStr, 0, 16);
10      ebp8 = 0;
11  l_4010AC:
12      if(ebp8 >= nameLen) {
13          ebp12 ^= 0xCAFE;
14          return (ebp12 == serialNum);
15      } else {
16          eax = ((unsigned int) nameStr) + ebp8;
17          edx = *((unsigned char *) eax) ^ (ebp8 + 0x90);
18          edx *= 0xABADF00D;
19          ebp12 ^= edx;
20          ecx = ebp8++;
21          goto l_4010AC;
22      }
23  }
```

Die verbleibende `goto`-Anweisung stellt offenbar eine Schleife dar, die über die Buchstaben des Namens iteriert. Sie wird deshalb als `for`-Schleife formuliert:

```
 1  int checkSerial(char *nameStr, char *serialStr) {
 2      unsigned int eax, ecx, edx, serialNum;
 3      int nameLen, i, correctSerial;
 4      nameLen = strlen(nameStr);
 5      correctSerial = 0;
 6      if(nameLen == 0) {
 7          return 0;
 8      }
 9      serialNum = strtoul(serialStr, 0, 16);
10      for(i = 0; i < nameLen; i++) {
11          eax = nameStr[i];
12          edx = ((char) eax) ^ (i + 0x90);
13          edx *= 0xABADF00D;
14          correctSerial ^= edx;
15      }
16      correctSerial ^= 0xCAFE;
17      return correctSerial == serialNum;
18  }
```

Der Angreifer könnte diese C-Funktion nun nutzen, um gültige Seriennummer zu erzeugen – dazu wird nach Zeile 17 der Wert von `correctSerial` ausgegeben. Für den Wert „Angreifer“ ergibt dies den hexadezimalen Wert 0xB1ED50E5, mit dem das Programm als Seriennummer „B1ED50E5“ erfolgreich registriert werden kann. Die statische Analyse hat dem Angreifer also die Möglichkeit gegeben, einen sogenannten *Keygen* für die Anwendung zu erzeugen, mit dem gültige Seriennummern für beliebige Namen erzeugt werden können, ohne dass das Programm dafür manipuliert werden muss. Zum Vergleich mit der tatsächlichen Funktion befindet sich der Quellcode der Beispielanwendung in Anhang B.1, die Funktion heißt dort `checkSerial`.

2.4.3 Dynamische Analyse

Das Vorgehen könnte durch eine dynamische Analyse noch vereinfacht werden: Schon ohne Umsetzung als C-Code ist ersichtlich, dass das Programm an Adresse 401104h einen Vergleich durchführt, der über den Erfolg der Prüfung der Seriennummer entscheidet. Der Angreifer kann nun einen Break Point mit einem Debugger wie beispielsweise OllyDbg [Yus13] auf diese Adresse setzen und erneut versuchen, das Programm mit der Eingabe des Namens „Angreifer“ zu registrieren. Dies wird den Break Point auslösen und dem Angreifer ermöglichen, den Inhalt der Register zu betrachten. Im Register `ECX` steht die gültige Seriennummer, wie Abbildung 11 zeigt.

Die dynamische Analyse wäre auch ohne vorangehende statische Analyse möglich – der Angreifer könnte den Break Point auf die API-Funktion `MessageBox` setzen, bevor er den OK-Button zur Registrierung betätigt, und würde danach auch ohne vorangehende statische Analyse das Umfeld des Maschinencodes finden, das die Fehlermeldung nach fehlgeschlagener Prüfung der Seriennummer zeigt.

In der Praxis ergänzt die dynamische Analyse oft die statische Analyse, denn mit ihr können Vermutungen, die im Rahmen einer statischen Analyse erhalten werden, schnell überprüft

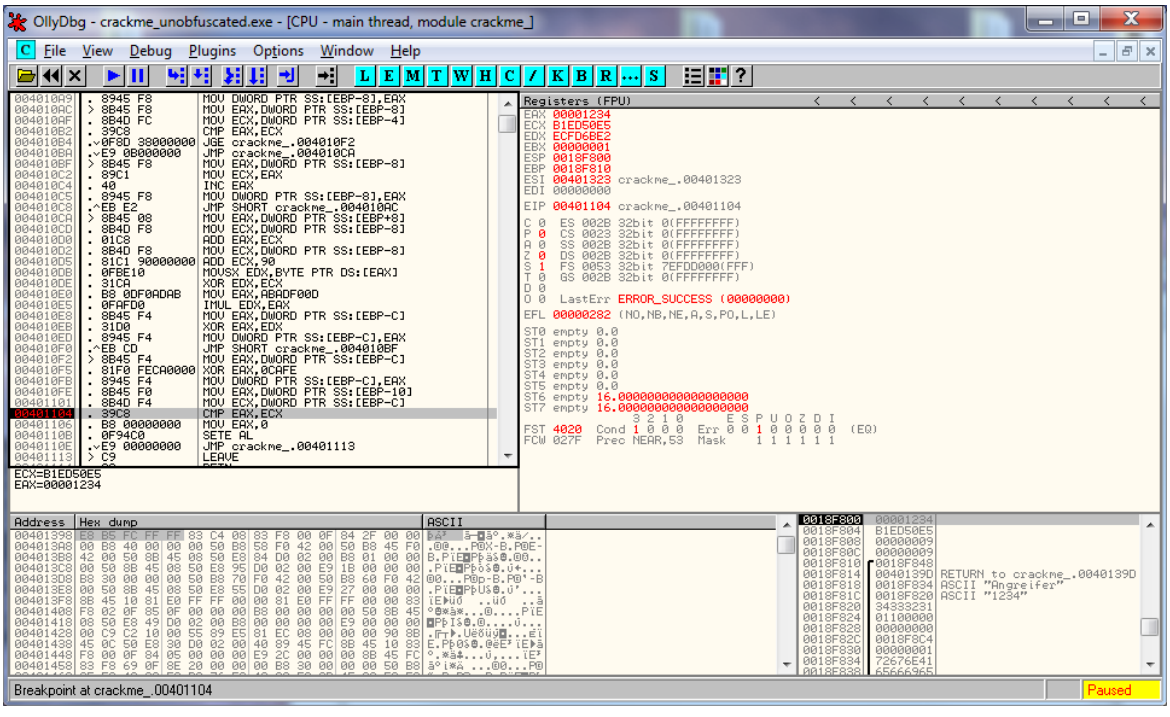


Abbildung 11: Auf eine beim Disassemblieren mit Kianxali entdeckte Stelle wurde ein Break Point im Debugger OllyDbg gesetzt. Es ist zu erkennen, dass im Register EAX die eingegebe, ausgedachte Seriennummer „1234“ geladen ist, die offenbar als hexadezimale Zahl 0x1234 interpretiert wurde. Im Register ECX befindet sich die korrekte Seriennummer 0xB1ED50E5, mit der verglichen wird.

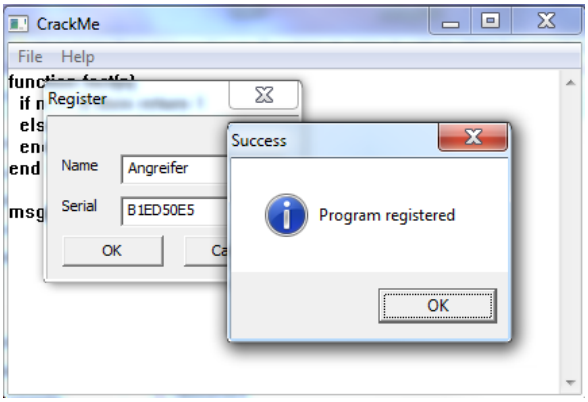


Abbildung 12: Die durch die Angriffe erhaltene Seriennummer funktioniert tatsächlich, wie eine erneutete Black-Box-Analyse des Programms bestätigt. Aus diesem Angriff wurde also eine korrekte Seriennummer für einen erfundenen Namen abgelesen. Diese Kombination aus Name und Seriennummer könnte ein Angreifer nun veröffentlichen.

werden. Dieses Vorgehen ist bei der Analyse von Schadsoftware allerdings riskant, da das Programm durch die dynamische Analyse zur Ausführung gelangt, was nicht immer gewünscht ist.

2.5 Zusammenfassung und Abgrenzung

Beide Analysemethoden haben zur Erzeugung einer gültigen Seriennummer geführt, mit der das Programm auch registriert werden kann (siehe Abbildung 12).

Die vorgestellten Angriffe waren unter anderem deshalb so einfach, weil die Funktion zur Überprüfung der Seriennummer zuerst die gültige Seriennummer berechnet und sie anschließend mit der eingegebenen vergleicht. Ein anderes Vorgehen wäre beispielsweise die Berechnung einer Eigenschaft von Name und Seriennummer und ein Vergleich diese Eigenschaften. Ein Beispiel wäre die Berechnung der Summe der ASCII-Werte von Name und Seriennummer und Prüfung auf Gleichheit. Auf diese Weise müsste der Angreifer die Funktion vollständig nachvollziehen, um eine gültige Seriennummer zu erzeugen. Wenn der Angreifer dazu nicht in der Lage ist, gibt es noch eine weitere Möglichkeit: Der Angreifer kann den Sprung, der über Korrektheit einer Seriennummer entscheidet, im Programm so manipulieren, dass *jede* Seriennummer gültig ist. In der Beispielanwendung könnte dazu der Sprung an der Adresse 4013A3h, der das Ergebnis des Funktionsaufrufes prüft, durch eine `nop`-Anweisung ersetzt werden. Das Programm würde dann jede Seriennummer als gültig akzeptieren und das Programm freischalten. Der Angreifer könnten einen sogenannten *Crack* für das Programm veröffentlichen. Dabei handelt es sich um ein Programm, das ein anderes Programm manipuliert. In diesem Fall würde es die entsprechenden Bytes manipulieren, um den Sprung durch eine `nop`-Anweisung zu ersetzen.

Folgende Eigenschaften des Programms haben dem Angreifer bei der Analyse und Umgehung des Kopierschutzes geholfen:

- da der Zugriff auf Zeichenketten eines Programms über deren Adressen erfolgt, sind Disassembler wie Kianxali in der Lage, für viele Zeichenketten die Stellen im Maschinencode anzuzeigen, an denen sie verwendet werden. So kann der Angreifer im Rahmen einer statischen Analyse anhand von Ausgaben des Programms schnell den dafür relevanten Code finden.

- die Aufrufe von API-Funktionen oder Funktionen aus dynamisch geladenen Bibliotheken werden in gängigen Betriebssystemen über den Namen der Funktion aufgelöst. Dies hilft dem Angreifer, da Disassembler wie Kianxali über eine Rückwärtsanalyse jeden Aufruf einer solchen Funktion mit dem Namen der Funktion anzeigen können. Wenn der Angreifer einen Aufruf wie `strlen` sieht, erfährt er darüber gleich mehrere Informationen: der Parameter ist eine Zeichenkette, der Rückgabewert eine Integer-Zahl.

- Schleifen können in Disassemblern anhand von Pfeilen leicht erkannt werden und liefern eine erste Orientierung bei der Analyse längerer Funktionen. Ein Disassembler kann diese Pfeile anzeigen, wenn der Kontrollflussgraph des Programms berechnet wurde.

In dieser Arbeit werden Verfahren zum Schutz vor Reverse Engineering vorgestellt, die ein Programm hauptsächlich vor statischen Analysen schützen. Ein Schutz vor dynamischen Analysen ist schwierig, da der Angreifer hierbei die Möglichkeit hat, das Programm bei seiner Ausführung zu beobachten, die Inhalte beliebiger Adressen des Programmspeichers zur Laufzeit auszulesen und mittels Break Points gezielt bestimmte Instruktionen im Maschinencode auf ihre Erreichbarkeit zu untersuchen. Allerdings ist eine dynamische Analyse für den Angreifer nur mit Einschränkungen möglich: Bereiche des Maschinencodes, die zur Laufzeit nicht erreicht werden, können nicht dynamisch analysiert werden. Die folgenden Beispiele verdeutlichen, welche Schwierigkeiten der Angreifer bei einer dynamischen Analyse haben kann:

- Zur Umgehung eines Kopierschutzes, der beispielsweise einen Datenträger auf Originalität prüft, wird der Angreifer einen originalen Datenträger besitzen müssen, um einen gültigen Lauf durch das Programm zu erhalten. Ansonsten werden bestimmte Prüfungen des Verfahrens nicht erreicht, da bereits die erste Prüfung bei fehlendem Datenträger fehlschlägt. Er könnte zwar möglicherweise alle Prüfungen umgehen, aber nicht den tatsächlichen Code der einzelnen Prüfungen in einer dynamischen Analyse nachvollziehen.

- Wenn ein Programm Informationen aus der Laufzeitumgebung verarbeitet und sich abhängig davon anders verhält, wird dies dem Angreifer bei einer dynamischen Analyse nicht auffallen, wenn nicht der gesamte Code manuell analysiert wird. So könnte ein Programm beispielsweise eine Funktion enthalten, die nur an einem bestimmten Kalenderdatum ausgeführt wird. Fällt dem Angreifer diese Prüfung in der dynamischen Analyse nicht auf, so bleibt ihm auch der Code verborgen, der an diesem Datum ausgeführt wird.

- Wenn das Programm über Netzwerkverbindungen mit anderen Programmen kommuniziert, beispielsweise mit einem Server des Herstellers, so wird der Angreifer nur die

Teile des Codes analysieren können, die zur Verarbeitung der aufgetretenen Netzwerknachrichten erforderlich waren. Falls der Hersteller bestimmte Netzwerkpakete nur selten verschickt, so wird dem Angreifer dies entgehen und der Code zur Verarbeitung dieser Nachrichten bleibt ihm verborgen.

In der Praxis wird dieses Wissen über die Angriffsmethoden von einigen Herstellern zum Schutz ihrer Programme angewendet: So wird eine Seriennummer zwar bei der Eingabe auf Gültigkeit geprüft und dies im Fehlerfall angezeigt. Bei dieser Prüfung werden allerdings nicht alle Eigenschaften geprüft, die eine gültige Seriennummer aufweisen muss. Ein Angreifer könnte diese Funktion analysieren und eine gültige Seriennummer erzeugen, die diese Prüfung besteht. Das Programm wird sich mit einer solchen Seriennummer auch zunächst wie erwartet verhalten, sodass der Angreifer seine Methode zur Erzeugung gültiger Seriennummern veröffentlicht. Allerdings wird das Programm zu einem späteren Zeitpunkt weitere Eigenschaften der Seriennummer prüfen, beispielsweise erst, sobald das Programm feststellen kann, dass es intensiv genutzt wird. Erst dann wird geprüft, ob die Seriennummer tatsächlich gültig ist und nicht nur die erste Prüfung bestanden hat. So kann der Hersteller nachweisen, dass die Nutzung illegal erfolgte. Erst jetzt wird der Angreifer erfahren, dass es eine weitere Prüfung im Code des Programms gab.

Die dynamische Analyse liefert also immer nur einen *partiellen* Blick auf das Programm, da nur ein oder mehrere Läufe des Programms betrachtet werden. Nur die statische Analyse kann einen Blick auf das gesamte Programm werfen und auch die Stellen im Code aufdecken, die bei einem bestimmten Lauf nicht erreicht werden. Aus diesem Grund wird die restliche Arbeit nur die statische Analyse diskutieren.

3 Obfuscation: Überblick

Insbesondere bei Anbietern, die Software kommerziell vertreiben, besteht oft der Wunsch, Produkte vor Reverse Engineering zu schützen. Der vorige Abschnitt hat gezeigt, dass Angreifer mit einfachen Mitteln den Kopierschutz einer Software entfernen oder Algorithmen der Software rekonstruieren können, wenn diese nicht dagegen geschützt ist. Einen Ansatz für einen solchen Schutz stellt *Obfuscation* dar – darunter wird eine Verschleierung des Maschinencodes verstanden, die dem Angreifer Hindernisse bei der Durchführung eines Angriffs bieten soll. In der Praxis existieren hierzu verschiedene Produkte in Form von Programmen, die ein Hersteller in den Übersetzungsprozess seiner Software integrieren kann, um diese ohne weitere Modifikationen automatisiert gegen Reverse Engineering zu schützen.

Neben der Verschleierung von Maschinencode gibt es auch Programme, die den Quellcode einer Software verschleiern – beispielsweise durch Umbenennung aller Variablen in zufällige Bezeichner, dem Entfernen sämtlicher Formatierung und ähnlichen Transformationen. Dies wird in der Regel nur bei interpretierten Sprachen wie JavaScript verwendet, wo Algorithmen im Quelltext an eine Software des Anwenders übertragen werden und geschützt werden sollen. In dieser Arbeit ist mit Obfuscation immer die Verschleierung von Maschinencode gemeint.

3.1 Einführung

Viele Veröffentlichungen zum Thema Obfuscation führen die Verschleierung als textuelle Transformation eines Assembler-Codes durch. Dieser wird erhalten, indem der Compiler für die Quellsprache angewiesen wird, keine ausführbare Datei, sondern ein Assembler-Listing zu erzeugen. Beim Compiler `gcc` ist dies beispielsweise über die Option `-S` möglich. Auf diesem Assembler-Listing findet nun die eigentliche Verschleierung statt, indem je nach Art der Verschleierung bestimmte Assembler-Konstrukte durch andere ersetzt werden. Das modifizierte Listing kann dann vom Compiler in eine ausführbare Datei übersetzt werden, die dann das verschleierte Maschinenprogramm darstellt. Eine wichtige Beobachtung bei dieser Vorgehensweise ist, dass der Programmierer keinen Einfluss darauf hat, welche Bereiche des Codes verschleiert werden. Die textuelle Transformation findet hier auf der gesamten Übersetzungseinheit statt.

Ein *Obfuscator* nach diesem Ansatz ist also ein spezieller Übersetzer $\mathcal{O}$ mit folgenden Eigenschaften:

- Quell- und Zielsprache sind Assembler
- Für übersetzte Assembler-Programme $P' = \mathcal{O}(P)$ gilt, dass für alle Eingaben $x_1, \ldots, x_n$ folgende Bedingung zutrifft:

  $$M(P, x_1, \ldots, x_n) = M(P', x_1, \ldots, x_n)$$

  Diese Bedingung sichert zu, dass durch die Verschleierung nicht die Semantik des Programms verändert wird – ein verschleiertes Programm soll sich zur Laufzeit identisch zur unverschleierten Variante verhalten.

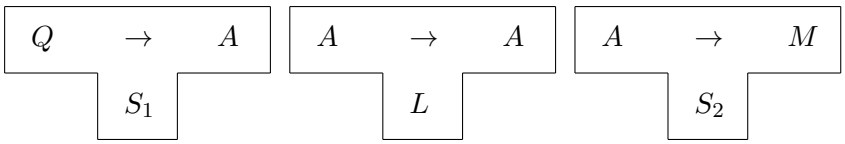


Abbildung 13: Das T-Diagramm zeigt den Ablauf der Verschleierung mithilfe eines Obfuscators: Ein in Sprache S_1 implementierter Compiler übersetzt die Quellsprache Q in einen Assembler-Dialekt A (links). Danach wird auf diesem Assembler-Listing die Verschleierung durch den in beliebiger Sprache L implementierten Obfuscator durchgeführt (mitte). Schließlich wird der verschleierte Assembler-Code einem in S_2 implementierten Assembler zugeführt, der ihn in Maschinensprache M übersetzt und dadurch ein verschleiertes Programm erzeugt. Oftmals verfügen Compiler über eine Möglichkeit zur Assemblierung, sodass der linke und rechte Teil des Diagramms dasselbe Programm sein können.

Dabei ist P ein als Assembler-Code vorliegendes Programm und P' die Ausgabe des Obfuscators, die als *verschleierter Code* bezeichnet wird. M ist ein Interpreter für den Assembler-Code.

Die Identitätsabbildung $\mathcal{O}(P) = P$ wäre also bereits ein trivialer Obfuscator. Ein sinnvoller Obfuscator sollte zusätzlich folgende Eigenschaften aufweisen [CT02, S. 738; BS05, S. 2]:

- Verschleierung: die White-Box-Analyse von P' sollte schwieriger als die White-Box-Analyse von P sein. Formal ist hiermit gemeint, dass bei Vorliegen einer Turingmaschine, die P' implementiert, keine Funktionswerte von P' automatisch und ohne Simulation der Turingmaschine berechnet werden können.
- Widerstandsfähigkeit: Es sollte schwierig sein, die Verschleierung automatisiert umzukehren
- Heimlichkeit: statistische Analysen von P' und P sollten sich wenig unterscheiden (z.B. relative Anzahl von Sprung-Anweisungen im Programm), um das Vorhandensein der Verschleierung zu verbergen
- Kosten: die Ausführungszeit und Programmgröße von P' und P sollten sich wenig unterscheiden

Abbildung 13 zeigt, wie ein Obfuscator als T-Diagramm dargestellt werden kann.

3.2 Unmöglichkeit von Obfuscation

Insbesondere für die erste Eigenschaft gibt es keine saubere Formalisierung, da es sich bei der White-Box-Analyse eines Programms um einen manuellen, kreativen Prozess des Angreifers handelt. Oft wird die Eigenschaft folgendermaßen formalisiert: Der Angreifer soll aus dem verschleierten Programm $\mathcal{O}(P)$ nur Eigenschaften berechnen können, die auch aus der reinen Beobachtung der Reaktion des Programms auf bestimmte Eingaben effizient berechenbar wären. Das Programm soll sich also trotz Vorliegen des verschleierten Maschinencodes

einer White-Box-Analyse entziehen und damit eine *Virtual Black Box* darstellen, bei der nur eine Black-Box-Analyse möglich ist. Es gibt ein theoretisches Ergebnis [Bar+12], in dem bewiesen wird, dass eine automatisierte Verschleierung nach dieser Formalisierung im Allgemeinen unmöglich ist. Im Beweis wird eine Familie von Einwegsfunktionen $\mathcal{P}$ mit folgenden Eigenschaften beschrieben:

- eine Black-Box-Analyse eines Programms $P \in \mathcal{P}$ erlaubt keine effiziente Rekonstruktion von P, d.h. aus der Beobachtung der Reaktion des Programms auf bestimmte Eingaben kann die implementierte Funktion nicht rekonstruiert werden

- aus einem beliebigen Programm P', das eine Funktion $P \in \mathcal{P}$ implementiert, kann effizient P rekonstruiert werden – der Zugriff auf das Maschinenprogramm erlaubt also die Rekonstruktion von P

Daraus folgt, dass Obfuscation nach der obigen Formalisierung nicht allgemein möglich ist, denn durch das Vorliegen eines verschleierten Programms $\mathcal{O}(P)$ kann wegen der zweiten Eigenschaft effizient P rekonstruiert werden, obwohl dies bei einer reinen Black-Box-Analyse wegen der ersten Eigenschaft nicht möglich wäre.

Die folgende Vereinfachung soll den Beweis veranschaulichen: Die Funktionen der Familie $\mathcal{P}$ bilden eine natürliche Zahl auf eine andere ab. Hierbei wird jeweils ein bestimmtes Bit der Eingabe ignoriert, sodass es keinen Einfluss auf den Funktionswert hat. Möchte ein Angreifer nun prüfen, um welche Funktion der Funktionsfamilie es sich handelt, so muss er ermitteln, welches Bit der Eingabe keinen Einfluss auf die Ausgabe hat. Dazu müssten aber mehrere Funktionswerte untersucht werden, um das irrelevante Bit über Vergleiche zu ermitteln. Bei hinreichender Langsamkeit des Programms wäre dieses Vorgehen ineffizient. In einer White-Box-Analyse hätte der Angreifer Zugriff auf den Maschinencode. Diesen könnte er in eine Turing-Maschine überführen und aus dieser wiederum einen äquivalenten Schaltkreis für die Funktion des Programms berechnen. In dieser könnte mittels einer einfachen Suche auf dem Schaltkreisgraph ermittelt werden, welche Eingabebits Einfluss auf die Ausgabebits nehmen können. Selbst bei beliebig komplexer Verschleierung des Programms wäre das irrelevante Eingabebit nicht mit der Ausgabe des Schaltkreises verknüpft, sodass die Überprüfung immernoch effizient möglich wäre.

In der Praxis findet Obfuscation dennoch Anwendung, da es bereits ein Vorteil ist, den Aufwand des Angreifers zu erhöhen, solange die Laufzeit des Programms nicht darunter leidet. Die praktischen Auswirkungen des obigen Ergebnisses sind noch nicht abschätzbar, da es sich auf eine sehr spezielle Klasse von Programmen bezieht. So ist Obfuscation zwar für den allgemeinen Fall unmöglich, aber es ist nicht bekannt, ob es Klassen von Programmen gibt, die Ausnahmen darstellen.

Einen anderen Ansatz zum Umgang mit diesem theoretischen Ergebnis zeigt Roberto Giacobazzi [GM12]: Die Möglichkeiten des Angreifers können im Rahmen eines realistischen Angreifermodells eingeschränkt werden, sodass eine Verschleierung sich nur an diesem Modell orientieren muss.

3.3 Einfache Transformationen zur Verschleierung

In Abschnitt 2.5 auf Seite 30 wurde beschrieben, welche Eigenschaften eines Programms dem Angreifer helfen, die für ihn interessanten Funktionen im Maschinencode zu lokalisieren und zu rekonstruieren. In diesem Abschnitt wird gezeigt, dass bereits einfache Transformationen den Angreifer an einer direkten Analyse hindern können. Anschließend werden die Transformationen hinsichtlich der oben vorgestellten Kriterien an Obfuscator untersucht.

3.3.1 Adresszugriffe

Die schnellste Form des Angriffs erfolgte über Zeichenketten, die im Zusammenhang mit der Funktion stehen. Ein Disassembler kann den Zugriff auf eine Zeichenkette innerhalb einer Funktion erkennen, da diese über ihre Adresse referenziert werden muss: da Zeichenketten üblicherweise Konstanten sind, werden diese im Datensegment des Programms gespeichert und über ihre Adresse im Codesegment referenziert. Der Aufruf einer Funktion zur Ausgabe einer Zeichenkette könne im Assembler-Code beispielsweise folgendermaßen aussehen:

```
1       push 402000h
2       call output
```

Ein Disassembler kann den Datentyp des Arguments nicht berechnen. In diesem Beispiel wäre es möglich, dass der Funktion die Integer-Konstante 0x402000 übergeben werden – oder dass die virtuelle Adresse 0x402000 zur Laufzeit die Adresse einer Zeichenkette ist. Moderne Disassembler wie Kianxali verfügen deshalb über eine Heuristik, die bei solchen Zugriffen prüft, ob die Konstante einer gültigen Adresse des virtuellen Adressraums entspricht und welche Art von Daten dort gespeichert sind. Durch eine einfache Transformation kann der Disassembler an dieser Analyse gehindert werden, wie das folgende Beispiel zeigt:

```
1  mov eax, 12747678h
2  xor eax, 12345678h
3  push eax
4  call output
```

Die Semantik ist äquivalent: Der ursprüngliche Code hat die Adresse der Zeichenkette (0x402000) direkt auf den Stack gelegt, während die verschleierte Variante die Adresse zur Laufzeit durch die exklusive Oder-Verknüpfung zweier Operanden berechnet und das Ergebnis auf den Stack legt: es gilt $0x12747678 \oplus 0x12345678 = 0x402000$. Der Effekt ist, dass der Disassembler nun nicht mehr erkennen kann, dass es sich bei dem Parameter um die Adresse einer Zeichenkette handeln könnte. Er kann dadurch nicht mehr diese Codestelle als Benutzung der Zeichenkette angeben. Dies ist problematisch für den Angreifer, da er zwar weiterhin die Zeichenkette im Datensegment des Programms vorfindet, aber nicht erfährt, an welchen Stellen des Maschinencodes sie referenziert wird.

3.3.2 Kontrollfluss

Wenn der Angreifer den Effekt einer Funktion nachvollziehen möchte, so ist insbesondere der Kontrollfluss der Funktion interessant: Sprünge vom Anfang zum Ende der Funktion entstehen häufig durch Gültigkeitsprüfungen der Funktionsparameter, bedingte Sprünge innerhalb der Funktion codieren Fallunterscheidungen und Sprünge entgegen dem Programmfluss gehören häufig zu Schleifen. Da grafische Disassembler wie Kianxali solche Sprünge durch Pfeile visualisieren, erhält der Angreifer so einen schnellen Überblick über interessante Teile einer Funktion. Auch dies kann durch einfache Transformationen erschwert werden. Hierzu können beispielsweise *opake Prädikate* (engl. opaque predicates) verwendet werden [Arb02]. Darunter werden boolesche Aussageformen verstanden, deren Wahrheitswert unabhängig vom Wert der Variablen ist. Beispielsweise ist die Aussageform „$7y^2 - 1$ ist keine Quadratzahl“ wahr für alle ganzen Zahlen. Zur Verschleierung kann dies genutzt werden, indem zur Laufzeit mit zufälligem y geprüft wird, ob die Aussage wahr ist (was immer der Fall ist). Über einen bedingten Sprung kann im Falsch-Fall beliebiger Code eingefügt werden, der zur Laufzeit nie erreicht wird, aber zur Verwirrung des Angreifers genutzt werden kann, falls diesem das genutzte opake Prädikat nicht bekannt ist. Die Funktion

$$f(x, y) = \begin{cases} 1 & \text{falls } 7y^2 - 1 = x^2 \\ 0 & \text{sonst} \end{cases}$$

liefert deshalb unabhängig von den übergebenen Parametern x und y das Ergebnis 0.

Der folgende Code zeigt eine beispielhafte Umsetzung:

```
1  mov eax, dword ptr[y]   ; y hierbei beliebige Adresse im Datensegment
2  imul eax, dword ptr[y] ; eax =  y^2
3  imul eax, 7            ; eax = 7y^2
4  sub eax, 1             ; eax = 7y^2 - 1
5  mov ebx, dword ptr[x]   ; x wieder zufaellige Adresse im Datensegment
6  imul ebx, dword ptr[x] ; ebx =  x^2
7  cmp eax, ebx           ; immer falsch, da eax keine Quadratzahl
8  jnz weiter             ; den folgenden Code zur Laufzeit immer
       ueberspringen
9  ...                    ; Hier Code einfuegen, der den Angreifer verwirrt
```

In diesem Code wird $7y^2 - 1$ mit dem Wert einer zur Übersetzungszeit zufällig gewählten Adresse im Datensegment berechnet, d.h. mit dem Wert einer globalen Variable, den der Angreifer zur Analysezeit möglicherweise nicht berechnen kann, wenn es mehrere Schreibzugriffe auf die Variable gibt und nicht bekannt ist, welcher Zugriff zuletzt erfolgte. Ebenfalls wird x^2 mit einem ebenfalls zufällig gewähltem x berechnet. Da der erste Ausdruck nie eine Quadratzahl sein kann, ist der Vergleich in Zeile 7 immer falsch, sodass auf diese Weise Kanten in den Kontrollflussgraphen des Programms eingefügt werden können, die zur Laufzeit nie erreicht werden. So können beispielsweise Schleifen und Abbruchbedingungen in Funktionen eingefügt werden, die zur Laufzeit übersprungen werden. Da ein Disassembler mit partieller Auswertung den Wahrheitswert solcher Prädikate nur prüfen kann, wenn ihm alle beteiligten Werte bekannt sind, kann dieser bei zufällig gewählten Adressen zur indirekten Adressierung

der Variablen keine Prüfung vornehmen und nicht ermitteln, dass die betroffenen Kanten des Kontrollflussgraphs unmöglich zur Laufzeit erreicht werden können.

3.3.3 Junk-Bytes

Der vorige Abschnitt hat eine Möglichkeit gezeigt, Stellen im Maschinencode zu erzeugen, die zur Laufzeit nie erreicht werden, da ein bedingter Sprung sie überspringt. Dies kann neben der Verwirrung des Angreifers auch zur Verwirrung eines Disassemblers genutzt werden. Der Disassembler kann nicht prüfen, ob ein Ausdruck ein opakes Prädikat ist, d.h. unabhängig vom Wert der Variablen einen festen Wahrheitswert besitzt. Der Disassembler wird also beiden Zielen des Sprungs folgen und den Code dort analysieren. Dies kann ein Obfuscator nutzen, um dort ein Byte einzufügen, das dem Anfang eines längeren Maschinencode-Befehls entspricht. Dieses Byte wird als Junk-Byte bezeichnet [SH12, S. 335]. Dadurch werden die dem Junk-Byte folgenden Bytes vom Disassembler als Teil der zum Junk-Byte gehörigen Anweisung analysiert, obwohl sie zu einer anderen Instruktion des anderen Sprungpfades gehören. Das folgende Beispiel verdeutlicht dies:

```
1  ...                  ; Berechnung des Praedikats wie oben
2  cmp eax, ebx         ; immer falsch, da eax keine Quadratzahl
3  jnz weiter           ; den folgenden Code zur Laufzeit immer ueberspringen
4  <Byte E8>            ; Byte E8, dem eigentlich 4 Bytes folgen (jmp-Befehl)
5  weiter:
6  ...                  ; Normaler Code der Anwendung
```

Das Byte mit dem hexadezimalen Wert 0xE8, das zur Laufzeit immer übersprungen wird, gehört in der x86-Architektur zu einer Sprunganweisung, die nach dem Byte noch 4 weitere Bytes mit dem Sprungziel erwartet. Die folgenden Bytes gehören aber bereits zu dem Teil des Codes, der auf den übersprungen Code folgt – der Disassembler ist also im Konflikt, ob die nächsten 4 Bytes dort zum ersten oder zweiten Pfad der Sprunganweisung gehören. Das Verhalten des Disassemblers ist hier von dessen Implementierung abhängig. Kianxali zeigt die Adresse des Codes im Fehlerprotokoll, damit die Stelle manuell betrachtet werden kann. Im schlimmsten Fall arbeitet der Disassembler nach dem Junk-Byte nicht mehr synchron zum tatsächlichen Code, sodass alle folgenden Anweisungen fehlerhaft disassembliert werden.

3.3.4 Arithmetische Verschleierungen

Bei der Verschleierung von arithmetischen Algorithmen ist oft gewünscht, dass der Angreifer nicht nachvollziehen kann, auf welche Weise die Ausgabe einer Funktion berechnet wird. Zu diesem Zweck gibt es viele Transformationen, die arithmetische Operationen durch andere ersetzen, die semantisch äquivalent sind, aber deren Effekt nicht offensichtlich ist. Eine Vielzahl solcher Transformationen findet sich im Buch „Hacker's Delight“ [War02], das eigentlich die Optimierung arithmetischer Ausdrücke beschreibt. Die Art der Optimierung findet allerdings auf so tiefer Maschinenebene statt, dass dadurch gleichzeitig eine Verschleierung gegeben ist. Eine bekannte Optimierung dieser Art ist beispielsweise, dass eine Multiplikation mit einer Zweierpotenz durch eine Bit-Schiebe-Operation ausgedrückt werden kann. Weniger bekannt

ist allerdings, dass der folgende Code den Wert des Registers EDI durch 7 teilt, das Ergebnis im Register EAX speichert und dabei schneller als der Divisionsbefehl des Prozessors ist (Beweis und Erläuterung im Buch auf Seite 189):

```
1  mov    ecx, 92492493h
2  mov    eax, edi
3  imul   eax, ecx
4  add    edx, edi
5  mov    ecx, edx
6  shr    ecx, 1
7  sar    edx, 2
8  mov    eax, edx
9  add    eax, ecx
```

Im Buch sind weitere Transformationen dieser Art beschrieben, die den Code schneller und gleichzeitig weniger nachvollziehbar machen. Ein Angreifer kann solchen Code zwar weiterhin in einer Hochsprache rekonstruieren, das Verständnis wird allerdings erheblich erschwert. Arithmetische Verschleierung ist daher besonders zum Schutz von proprietären Algorithmen geeignet, bei denen der Angreifer nicht erfahren soll, wie oder warum eine bestimmte Funktion im Programm funktioniert.

3.3.5 Überlappender Code

Der vorletzte Abschnitt hat gezeigt, wie Code erzeugt werden kann, der aus Sicht des Disassemblers zu zwei verschiedenen Anweisungen gehört, obwohl zur Laufzeit nur eine der Anweisungen ausgewertet wird. Es gibt allerdings auch Erweiterungen dieser Technik, bei denen bestimmte Bytes zu mehreren Instruktionen gleichzeitig gehören und auch alle davon zur Laufzeit ausgeführt werden [Kin10, S. 27]. Dies ist sowohl für den Disassembler als auch für den Angreifer nur schwierig nachzuvollziehen, wie folgendes Beispiel zeigt:

```
1  401000 66B8EB05        mov ax, 05EBh
2  401004 31C0            xor eax, eax
3  401006 74FA            jz 4010002
4  401008 E890909090      call 90D0A09Dh
```

Bei den Zeilen 2 und 3 handelt es sich um ein konstantes Prädikat – die exklusive Oder-Verknüpfung identischer Operanden liefert immer 0, sodass der Sprung in Zeile 3 immer ausgeführt wird. Auffällig ist, dass das Sprungziel *innerhalb* der Anweisung von Zeile 1 liegt. Die `call`-Anweisung in Zeile 4 wird also zur Laufzeit nie erreicht, stattdessen wird den Sprung in Zeile 3 folgender Code ausgeführt, der nur einer anderen Sicht auf dieselbe Bytefolge entspricht:

```
1  401002 EB05            jmp 401009
2  401004 31C0            xor eax, eax
3  401006 74FA            jz 4010002
4  401008 E8              ; junk-byte
5  401009 90909090        nop nop nop nop
```

Die `mov`-Anweisung in Zeile 1 des vorletzten Listings wurde zwar zur Laufzeit ausgeführt, hatte aber tatsächlich die Aufgabe, eine Sprunganweisung mit den Bytes EB 05 im Parameter

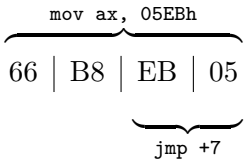


Abbildung 14: Die Abbildung zeigt, wie eine Maschinencode-Anweisung innerhalb einer anderen versteckt werden kann. Die 4 dargestellten Bytes entsprechen dem Opcode `mov ax, 05EBh`, der eine Konstante in ein Register kopiert. Werden die ersten zwei Bytes des Befehls zur Laufzeit allerdings übersprungen, so wird die innerhalb der Anweisung versteckte Sprunganweisung `jmp +7` ausgeführt, die um 7 Bytes nach vorn springt. Das Überspringen der ersten beiden Bytes geschieht zur Laufzeit durch einen Sprung in die Mitte der Anweisungsbytes.

zu verstecken (siehe Abbildung 14). Diese wird nun ausgeführt, sodass in Zeile 5 gesprungen wird. Der Parameter der `call`-Anweisung in Zeile 4 des vorletzten Listings diente also ebenfalls zur Verschleierung eines Befehls, der nun ausgeführt wird. In diesem Beispiel sind es 4 `nop`-Anweisungen. Auf diese Weise können Befehle mit kurzen Bytefolgen vollständig vor dem Disassembler und somit auch vor dem Angreifer versteckt werden. Der einzige Anhaltspunkt ist der Sprung in Zeile 3 des vorletzten Listings, da das Sprungziel innerhalb einer bereits disassemblierten Anweisung liegt. Kianxali würde dies durch eine Warnung im Protokoll anzeigen.

3.3.6 Bewertung

Die vorstellten Verfahren lassen sich einfach als textuelle Musterersetzung implementieren, da sie keine Kenntnis über die Semantik des Umfelds benötigen. Es wurde argumentiert, warum die Verfahren die White-Box-Analyse erschweren. Der Einfluss auf die Laufzeitgeschwindigkeit des verschleierten Programms ist unterschiedlich – er hängt nicht nur davon ab, wie viel größer das ersetzte Muster als das gesuchte Muster ist. Moderne Prozessoren besitzen dank Techniken wie Pipelining und Sprungvorhersagen weitreichende Möglichkeiten zur optimierten Ausführung von Maschinencode. Insbesondere das Einfügen von Verzweigungen ist daher langsam, da hierdurch oft die Pipeline des Prozessors geleert werden muss. Eine Analyse der tatsächlichen Laufzeitunterschiede kann daher nur durch Messungen erfolgen. An dieser Stelle sollen jedoch hauptsächlich die Kriterien Heimlichkeit und Widerstandsfähigkeit untersucht werden.

Heimlichkeit Wenn die Verfahren wie oben beschrieben als textuelle Transformation von Assembler-Code durchgeführt werden, wird sich eine statistische Analyse des verschleierten Programms signifikant von unverschleierten Maschinenprogrammen unterscheiden, die mit demselben Compiler übersetzt wurden. Das vorstellte Verfahren zur Verschleierung von Zugriffen auf Adressen ersetzt beispielsweise alle Vorkommen des Befehls `push <zahl>` durch eine Laufzeitberechnung der Zahl gefolgt von `push <register>`. Dadurch taucht die Variante des `push`-Befehls, die ein Literal auf den Stack legt, im verschleierten Programm nicht mehr auf.

Bei einem unverschleierten Programm wäre das Fehlen dieses Musters sehr unwahrscheinlich – ein solches Programm könnte keine Funktionen mit Literalen als Parameter aufrufen.

Das Vorhandensein Junk-Bytes und überlappende Instruktionen lässt sich durch einen Disassembler automatisiert prüfen, da dieser während des Disassemblierens erkennt, dass ein Byte zu zwei Instruktionen gehört oder eine ungültige Instruktion ergibt. Kianxali zeigt solche Programmstellen in der Fehlerkonsole an. Arithmetische Verschleierungen und das Einfügen von opaken Prädikaten zur Verschleierung des Kontrollflusses lassen sich im Allgemeinen nicht automatisiert aufdecken. Der Compiler LLVM generiert beispielsweise bei eingeschalteter Optimierung tatsächlich den oben vorgestellten Code zur Division durch 7, falls im Code eine Division durch eine konstante Ganzzahl geschieht. Es lässt sich daher nicht unterscheiden, ob der Compiler den Code optimiert hat oder arithmetische Operationen verschleiert wurden. Opake Prädikate lassen sich nur aufdecken, wenn diese dem Angreifer bekannt sind.

Widerstandsfähigkeit Sobald der Angreifer erkennt, dass eines der oben vorgestellten Verfahren genutzt wurde, kann er die Transformationen effizient wieder umkehren. In Kianxali gibt es dazu die Möglichkeit, über eine Ruby-Schnittstelle mit dem disassemblierten Code zu interagieren und Transformationen vorzunehmen. So könnte die Transformation zur Verschleierung von Adresszugriffen umgekehrt werden, indem nach dem Muster `mov <reg>, <zahl>; xor <reg>, <zahl>; push <reg>` gesucht wird, die exklusiv Oder-Verknüpfung in Ruby durchgeführt und die Befehlsfolge durch `push <ergebnis>` ersetzt wird. Dadurch wird der ursprüngliche Code rekonstruiert, die Verschleierung also umgekehrt. Ebenfalls können Junk-Bytes entdeckt und durch eine effektlose `nop`-Anweisung ersetzt werden. Überlappender Code kann mit Kianxali zwar entdeckt, aber nicht in unüberlappenden Code umgewandelt werden, da es auch in Kianxali die Einschränkung gibt, dass jedes Byte nur zu einem Befehl gehören kann. Hierzu gibt es allerdings andere Ansätze [Kin10].

Fazit Die Verwendung einfacher Transformationen erscheint wegen der einfachen Umsetzung verlockend, zumal es plausibel erscheint, warum die Analyse für den Angreifer erschwert wird. Problematisch wird dieser Ansatz jedoch, wenn die Transformationen auf dem gesamten Programm durchgeführt werden, denn dadurch wird dem Angreifer schnell auffallen, dass das Programm verschleiert ist – beispielsweise, wenn im Programm viele Zeichenketten vorhanden sind, aber keine Referenz auf diese im Maschinencode auftaucht. Bei der weiteren Durchsicht des Assembler-Codes kann der Angreifer die Transformationen an vielen Stellen erkennen und schließlich ein Skript zur automatisierten Umkehrung entwerfen. Wenn hingegen nur eine einzelne Zeichenkette mit einer solchen Transformation geschützt wäre, könnte der Angreifer weder die Referenz auf die Zeichenkette, noch den transformierten Zugriff auf diese mit einfachen Mitteln auffinden. Beim Entwurf von Transformationen ist also stets zu bedenken, ob die Kenntnis des Verfahrens dem Angreifer ein Umkehren ermöglichen würde, ansonsten handelt es sich um *security by obscurity*. Eine weitere Möglichkeit wäre, die Anwendung der Transformation nicht auf das gesamte Programm anzuwenden. Dies ist beim Ansatz eines externen Obfuscators, der auf dem Assembler-Listing eines Compilers arbeitet, aber nur schwierig möglich, da hier die gleichen Probleme entstehen, die ein Disassembler bei der Analyse des Listings hätte.

Im Folgenden werden zwei Verfahren beschrieben, die sich auch mittels partieller Auswertung durch einen Disassembler nicht umkehren lassen, sondern zur Umkehrung Wissen über das Verfahren benötigen.

3.4 Selbstmodifizierender Code

Unter selbstmodifizierendem Code werden Anweisungen verstanden, die schreibend auf das Codesegment zugreifen, also Maschinencode an anderer Stelle zur Laufzeit modifizieren. Das folgende Beispiel zeigt einen beispielhaften Ausschnitt selbstmodifizierenden Codes aus einem modernen Obfuscation-Verfahren von Balachandran [BE13]:

```
1  40102B 83353210400051 xor byte ptr [401032h], 51h
2  401032 B805000000     mov eax, 5
3  401037 B901000000     mov ecx, 1
```

Die Anweisung in Zeile 1 führt eine exklusive Oder-Verknüpfung mit dem ersten Byte der Anweisung in Zeile 2 durch. Dadurch ändert sich das Byte 0xB8 an der Stelle in $0xB8 \oplus 0x51 = 0xE9$. Die `mov`-Anweisung wird dadurch *zur Laufzeit* in eine `jmp`-Anweisung geändert, die einen relativen Sprung um 5 Bytes ab dem Ende der Instruktion durchführt, wodurch die folgende `mov`-Anweisung in Zeile 3 genau übersprungen wird. Aus Sicht des Disassemblers werden sowohl Zeile 2 als auch Zeile 3 als `mov`-Anweisung interpretiert, die im Kontrollflussgraph auch direkt miteinander verbunden sind, also nacheinander ausgeführt werden. Zur Laufzeit wird aber keine der Anweisungen in dieser Form ausgeführt, stattdessen sorgt der erste Befehl dafür, dass der dritte übersprungen wird. Es werden also weder das Register EAX noch ECX verändert. Dies ist für den Angreifer auf den ersten Blick nicht sichtbar und wird von automatisierten Analysen nicht entdeckt. Zum Aufdecken des Verfahrens müsste der Disassembler erkennen, dass die Anweisung in Zeile 1 eine Änderung im Code-Segment durchführt. Kianxali unterstützt dies, sodass auch dieses Verfahren automatisiert umgekehrt werden kann. Das Skript in Anhang B.3 auf Seite 104 zeigt, wie die oben gezeigte Stelle automatisch aufgedeckt und zurücktransformiert werden kann.

Neben der geringen Heimlichkeit hat das Verfahren aber weitere Nachteile. Sobald die Sprunganweisung durch den selbstmodifizierenden Code umgewandelt wurde, kann diese Stelle des Codes nicht noch einmal ausgeführt werden. Dann würde die exklusive Oder-Verknüpfung erneut ausgeführt und die Sprunganweisung würde wieder in die ursprünglich codierte `mov`-Anweisung überführt, wodurch die Semantik des Programms verletzt wäre, da der Sprung dann nicht ausgeführt würde. Zur Lösung fügt Balachandran bei jedem Sprungziel erneut selbstmodifizierenden Code ein, der den `mov`-Befehl wiederherstellt. Dies funktioniert aber nur fehlerfrei, wenn der Bereich nicht durch nebenläufige Threads erreicht werden kann. Andernfalls gäbe es einen Lauf, in dem ein Thread die Sprunganweidung decodiert, dem Sprung folgt und gleichzeitig ein anderer Thread sie wieder zu einer `mov`-Anweidung codiert. Nachdem der erste Thread das Sprungziel erreicht hat, würde er diese sofort wieder zu einer `jmp`-Anweisung ändern, sodass eine erneute Ausführung eines beliebigen Threads die Stelle im falschen Zustand erreicht. Weiterhin kann das Verfahren keine indirekten Sprünge transformieren, bei denen sich das Sprungziel in einem Register befindet. Der Grund ist, dass

der Obfuscator in diesem Fall nicht berechnen kann, welche Ziele bei diesem Sprung möglich sind, um dort den modifizierten Code wiederherzustellen.

3.5 Opake Konstanten

Die bereits diskutierten opaken Prädikate können nicht automatisiert aufgedeckt werden, da hierzu Instanzen des Entscheidungsproblem für die Menge der Tautologien berechnet werden müssten – dieses Problem ist aber innerhalb der Prädikatenlogik nicht berechenbar. Ein Verfahren von Moser, Kruegel und Kirda [MKK07] erweitert diesen Ansatz, um nicht nur boolesche Werte zu verschleiern, sondern beliebige Integer-Konstanten. Dazu wird die Binärdarstellung $b_0 b_1 b_2 \dots b_n$ der zu verschleiernden Konstante mithilfe von opaken Prädikaten zur Laufzeit berechnet. Es werden also n opake Prädikate benötigt, wobei das Prädikat P_i jeweils den Wahrheitswert b_i besitzen muss.

Im oben erwähnten Verfahren werden die nötigen Prädikate mithilfe zufällig erzeugter aussagenlogischer Formeln erzeugt. Die Sicherheit der Prädikate basiert darauf, dass das 3-SAT-Problem NP-vollständig ist.

3.5.1 3-SAT

Die Menge 3-SAT enthält alle erfüllbaren aussagenlogischen Formeln, die in konjunktiver Normalform sind und höchstens 3 Literale pro Klausel enthalten. Eine solche Formel ist beispielsweise

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_3)$$

Die obige Formel ist beispielsweise mit $x_3 = 1$ und beliebigen Werten für die anderen Variablen erfüllbar. Die folgenden Formel ist nicht in 3-SAT enthalten, da sie nicht erfüllbar ist:

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1}) \wedge (\overline{x_2}) \wedge (\overline{x_3})$$

Da es zu jeder aussagenlogischen Formel eine äquivalente aussagenlogische Formel in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel gibt, wird im Folgenden nur noch von „aussagenlogischen Formeln“ gesprochen, wenn „aussagenlogische Formeln in konjunktiver Normalform mit höchstens 3 Literalen pro Klauseln“ gemeint ist. Das Entscheidungsproblem, ob eine Formel in 3-SAT liegt oder nicht, ist NP-vollständig. Es ist also kein Verfahren bekannt, das effizient entscheidet, ob eine aussagenlogische Formel erfüllbar ist. Neben der Menge 3-SAT sind für dieses Verfahren noch zwei weitere Mengen relevant:

- 3-UNSAT: die Menge der Kontradiktionen, d.h. der nicht erfüllbaren Formeln.
- 3-TAUT: diese Teilmenge von 3-SAT enthält alle Tautologien, d.h. alle Formeln, die von *jeder* Belegung erfüllt werden.

Bei der Betrachtung einer beliebigen aussagenlogischen Formel kann nicht effizient entschieden werden, ob diese zu 3-UNSAT oder zu 3-SAT (und darin ggf. zu 3-TAUT) gehört. Es ist allerdings möglich, *zufällige* aussagenlogische Formeln zu erzeugen, die mit hoher Wahrscheinlichkeit zu einer vorher bestimmten Klasse gehört.

3.5.2 Zufällige Erzeugung aussagenlogischer Formeln

Selman, Mitchell und Levesque haben untersucht [SML96], welchen Einfluss das Verhältnis von der Anzahl der Klauseln und der Anzahl der Variablen bei der zufälligen Erzeugung von aussagenlogischen Formeln auf die Erfüllbarkeit der Formel hat. Dazu wurde ein SAT-Solver nach dem DPLL-Algorithmus mit zufällig erzeugen Formeln als Eingabe untersucht. Es wurden die Anzahl der Rekursionen des DPLL-Algorithmus und die Erfüllbarkeit der Formeln analysiert. Das Ergebnis ist interessant:

- wenn das Verhältnis von Klauseln zu Variablen kleiner als 4 ist, d.h. wenn es weniger als viermal so viele Klauseln wie Variablen gibt, ist die Wahrscheinlich dafür, dass die zufällig erzeugte Formel erfüllbar ist, sehr hoch. Die Wahrscheinlichkeit nimmt weiter zu, je kleiner das Verhältnis wird. Der SAT-Solver benötigt verhältnismäßig wenig Rekursionen, um eine solche Formel zu prüfen. Die Anzahl der Rekursionen fällt mit kleinerem Verhältnis steil ab.
- wenn das Verhältnis von Klauseln zu Variablen 4.25 beträgt, ist die Wahrscheinlichkeit für die Erfüllbarkeit der zufällig erzeugten Formel 50%. Gleichzeitig benötigt der SAT-Solver für diese Formeln die meisten Rekursionen.
- übersteigt das Verhältnis von Klauseln zu Variablen den Wert 5, so ist die Wahrscheinlich für die Erfüllbarkeit der zufällig erzeugten Formeln sehr klein. Sie sinkt weiter, je größer das Verhältnis wird. Die Anzahl der Rekursionen des SAT-Solvers sinkt auch hier wieder, allerdings viel flacher als im ersten Fall.

Das Ergebnis ist in Abbildung 15 visualisiert. Über das Verhältnis der Anzahl von Klauseln zu Variablen kann also sowohl die Schwierigkeit für einen SAT-Solver als auch die Erfüllbarkeit der Formel gesteuert werden. In einer späteren Arbeit wurden weitere Parameter untersucht, die Einfluss auf die Erfüllbarkeit und Schwierigkeit der Formeln haben – dort wird ein Modell vorgestellt, das die Erzeugung von sehr schwierigen 3-SAT-Instanzen bei gleichzeitig sehr hoher Wahrscheinlichkeit für die Erfüllbarkeit erlaubt [Xu+05]. Es ist auch möglich, erfüllbare Formeln mit vorgegebener Belegung zu erzeugen. Dazu wird bei der Erzeugung von zufälligen Klausen jede Klausel verworfen, die von der Belegung nicht erfüllt wird. Auf diese Art erzeugte Instanzen sind allerdings sehr schnell von einem SAT-Solver zu lösen [Xu+05].

3.5.3 Erzeugung verschleierter Konstanten

Mit diesem Wissen können nun beliebige Wahrheitswerte zur Laufzeit eines Programms erzeugt werden: Soll ein 0-Bit erzeugt werden, so wird während der Verschleierung eine zufällige, aber unerfüllbare 3-SAT-Instanz erzeugt, die als Klauselmenge im Datensegment der

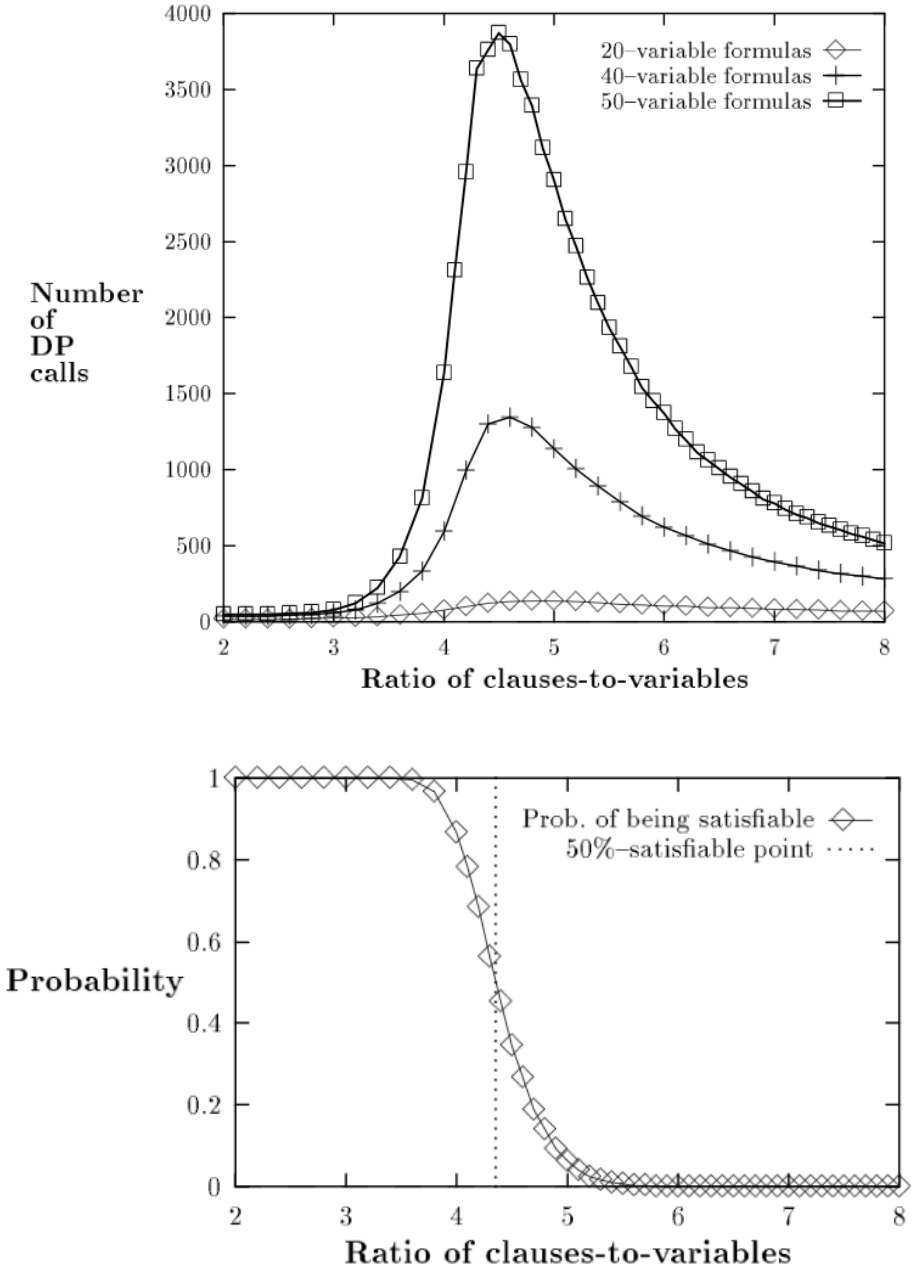


Abbildung 15: Über das Verhältnis von Klauseln zu Variablen kann bei der Erzeugung von zufälligen 3-SAT-Instanzen gesteuert werden, wie schwierig die Instanz mit dem DPLL-Algorithmus zu lösen ist (oben). Gleichzeitig bestimmt das Verhältnis die Wahrscheinlichkeit für die Erfüllbarkeit der Formel (unten). Bei einem Verhältnis von 4.25 ist die Wahrscheinlichkeit ist die Formel am schwierigsten mit dem DPLL-Algorithmus zu untersuchen, gleichzeitig beträgt die Wahrscheinlichkeit für die Lösbarkeit ca. 50%. Quelle: [SML96]

Anwendung gespeichert wird. Wenn das 0-Bit zur Laufzeit benötigt wird, so wird an der Stelle Code erzeugt, der die SAT-Instanz mit Variablen auswertet, die zur Laufzeit zufällig bestimmt werden, beispielsweise die Benutzung von beliebigen Werten im Datensegment, die einer ständigen Änderung während der Laufzeit unterliegen. Da das Ergebnis der SAT-Auswertung unabhängig von der Belegung der Variablen Null ist, wurde auf diese Weise ein 0-Bit erzeugt. Analog kann ein 1-Bit erzeugt werden, indem eine tautologische Formel durch Negation einer unerfüllbaren Formel erzeugt wird. Eine weitere Möglichkeit ist es, eine erfüllbare, nicht-tautologische Formel mit bekannter Erfüllbarkeitsbelegung zu erzeugen und die Variablen zur Laufzeit so zu wählen, dass die Formel erfüllt wird.

Um dieses Verfahren umzukehren, also die Konstanten ohne Simulation des Programms zu berechnen, müsste der Angreifer erkennen, ob eine gegebene SAT-Instanz zur Menge 3-SAT oder 3-UNSAT gehört. Dies ist bei kleinen Instanzen mit einem SAT-Solver praktisch möglich. Wenn die Instanzen so groß gewählt werden, dass ein Angriff via SAT-Solver zu aufwändig wird, wächst die Größe des verschleierten Programms allerdings stark an, da für jedes konstante Bit eine große SAT-Instanz im Datensegment abgelegt werden muss. Auch die Ausführungsgeschwindigkeit sinkt bei größeren Instanzen. Das Ziel der Autoren war allerdings nur das Aufzeigen, dass eine statische Analyse im Allgemeinen nicht ausreichen kann, um ein Programm zu analysieren. Ein Model-Checker, der Aussagen über ein auf diese Weise verschleiertes Programm treffen wollte, müsste für opake Konstanten einen sehr großen Zustandsraum durchsuchen. Durch Kenntnis des Verfahrens könnte ein SAT-Solver dies allerdings erheblich beschleunigen. Der Aufwand des Angreifers (Extraktion der SAT-Instanzen, Analyse mit einem SAT-Solver, Rekonstruktion der Konstanten) ist trotzdem hoch.

3.6 Diskussion

In diesem Abschnitt wurden verschiedene Verfahren zur Verschleierung von Programmen vorgestellt. Keines der Verfahren vermag einen Angreifer vollständig aufzuhalten, allerdings können einige der gezeigten Verfahren den Aufwand im Vergleich zu einem unverschleierten Programm merklich erhöhen. Insbesondere hat sich gezeigt, dass die Kenntnis des Verfahrens dem Angreifer einen starken Vorteil verschafft – anders als beispielsweise im Bereich der Kryptographie, bei der es als selbstverständlich gilt, Verfahren zu veröffentlichen und ihnen trotzdem (oder gerade deshalb) zu vertrauen. Zur Verbesserung der Situation werden also entweder neue Verfahren benötigt oder die bestehenden müssen auf eine Weise angewendet werden, bei der ein Angreifer das Vorhandensein der Verschleierung gar nicht erst bemerkt (vgl. dazu das Fazit in Abschnitt 3.3.6 auf Seite 40).

4 Ein neuer Ansatz für Obfuscation

Der vorige Abschnitt hat gezeigt, dass einfache Transformationen nur einen Schutz bieten können, wenn sie nur auf kleine Bereiche des Programms angewendet werden. Wird das ganze Programm transformiert, so fällt dies durch statistische Analysen auf und ermöglicht es dem Angreifer, verschleierte Bereiche zu finden, die Verschleierung zu analysieren und schließlich ein Skript zur Umkehrung der Transformation zu entwerfen. Bestimmte Techniken wie Junk-Bytes und überlappende Instruktionen zeigen dem Angreifer bei der Verwendung eines modernen Disassemblers sogar, wo sich die verschleierten Stellen im Programm befinden. Von solchen Techniken sollte daher abgesehen werden.

Eine selektive Anwendung von Verschleierung ist jedoch schwierig, da ein Obfuscator nicht die Semantik der Hochsprachenkonstrukte kennt, die zur Erzeugung einer bestimmten Anweisung im Assembler-Listing des Compilers geführt haben. Im Listing fehlen ebenfalls die Informationen über die beteiligten Datentypen von Berechnungen, da es auf Assembler-Ebene keine Datentypen mehr gibt. Der bisherige Ansatz, für Obfuscation einen externen Obfuscator in den Ablauf des Compilers zu integrieren, soll daher insgesamt überdacht werden. Er ermöglicht zwar eine rasche Implementierung neuer Verfahren, da oft nur textuelle Transformationen durchgeführt werden müssen – die vorgestellten Nachteile erscheinen aber so gravierend, dass nun ein neuer Ansatz für Obfuscation vorgestellt wird. Dazu soll die Möglichkeit zur Verschleierung direkt in einen Compiler integriert werden, wodurch folgende Vorteile erwartet werden:

- selektive Verschleierung: Der Programmierer kann durch Annotationen im Hochsprachencode genau festlegen, welche Bereiche des Quellcodes verschleiert werden sollen. Wenn mehrere Verfahren implementiert werden, kann er über die Annotationen bestimmen, welche Verschleierungstechnik auf welche Bereiche des Codes angewandt werden sollen. Auf diese Weise wird die Heimlichkeit der Verschleierung erhöht – wenn nur ein kleiner Teil des Programms verschleiert ist, kann der Angreifer sie im besten Fall nicht finden und bemerkt daher nicht, dass ein Teil des Programms verschleiert ist.

- Kenntnis aller Datentypen: während auf Assembler-Ebene nicht mehr unterschieden werden kann, welchen Datentyp eine Zahl in einem Register hat, sind dem Compiler alle Datentypen der beteiligten Variablen bekannt. Dies ist vorteilhaft, wenn beispielsweise nur arithmetische Ausdrücke verschleiert werden sollen, nicht aber Zeiger-Arithmetik auf Adressen. Ebenfalls ist dem Compiler bekannt, ob Variablen als konstant markiert wurden – dies kann für Verschleierung wichtig sein, weil dann bekannt ist, dass keine schreibenden Zugriffe auf eine Variable stattfinden werden. Im Assembler-Listing ist dies ebenfalls nicht erkennbar.

- Kenntnis von Funktionssignaturen: Dem Compiler ist zu jeder Funktion deren Signatur bekannt. Dadurch kann sichergestellt werden, ob es sich bei einem Parameter um die Adresse einer Zeichenkette, eine Integer-Zahl oder einer dynamischen Datenstruktur handelt. Dies ist vorteilhaft, wenn eine Verschleierung beispielsweise nur Zeichenketten, nicht aber andere Datenstrukturen betreffen soll.

- Verschleierung vor Optimierung: Wird auf dem Assembler-Listing eines Compilers gearbeitet, so ist der Code dort bereits vollständig optimiert. Durch das Inlining von Funktionen oder die Entrollung von Schleifen ist der Kontrollfluss der Hochsprachenkonstrukte nicht mehr vollständig zu erkennen. Dies ist bei einigen Techniken nachteilig, zum Beispiel wenn arithmetische Ausdrücke verschleiert werden sollen, diese allerdings durch den Compiler bereits so optimiert werden, dass sie nicht mehr als solche zu erkennen sind.
- Kenntnisse von Funktionszeigern: Anweisungen wie `call eax` können von einem externen Obfuscator-Programm nicht transformiert werden, da das Ziel des Aufrufs unbekannt ist. Ein Compiler kennt die möglichen Ziele allerdings, sofern der Programmierer dieses nicht zur Laufzeit über Zeigerarithmetik berechnen lässt. Dadurch können auch diese Fälle verschleiert werden.

Es gibt bereits eine bestehendes Projekt, das ebenfalls eine Integration von Obfuscation in einen Compiler anstrebt [Vau14]. Allerdings können die Verschleierungen auch hier nur auf die gesamte Übersetzungseinheit angewendet werden, während die oben vorgestellte Idee eine durch Annotationen sehr lokale Anwendung von Verschleierung ermöglicht. Zudem werden die Verschleierungen im bestehenden Projekt auf der Ebene des Zwischencodes des Compilers durchgeführt, sodass kein Wissen über die Zielarchitektur angewendet werden kann, welches beispielsweise zur Umsetzung von überlappenden Code nötig wäre.

Zur Evaluation der Idee werden zunächst Verfahren zur Integration in einen Compiler diskutiert, die sich gut für die lokale Anwendung mithilfe von Annotationen eignen.

4.1 Verbesserung des Verfahrens von Balachandran

In Abschnitt 3.4 auf Seite 42 wurde das Verfahren von Balachandran vorgestellt, das Sprünge durch selbstmodifizierenden Code tarnt. Mit diesem Verfahren können alle Sprünge einer Funktion so verschleiert werden, dass der Disassembler keine Verzweigungen im Kontrollflussgraph der Funktion entdeckt, da die Sprünge erst unmittelbar vor ihrer Ausführung zur Laufzeit durch selbstmodifizierenden Code zu Sprunganweisungen umgeschrieben werden. Der Nachteil dieses Ansatzes ist, dass selbstmodifizierender Code schnell durch den Angreifer entdeckt werden kann, sodass gerade die eigentlich verschleierten Stellen auffallen. Die überlappenden Anweisungen aus Abschnitt 3.3.5 hatten den gleichen Nachteil, obwohl die Idee viel Potenzial zum Verstecken von Maschinencode-Anweisungen bietet. Die beiden Verfahren werden nun auf eine Weise kombiniert, die das Aufdecken erschwert und gleichzeitig weitere Nachteile des Verfahrens von Balachandran beseitigt.

Die Idee hierzu ist, dass überlappende Instruktionen genutzt werden, ohne dass ein direkter Sprung von einer Instruktion in die Bytefolge einer anderen Instruktion erfolgt. Auf diese Weise kann ein Disassembler nicht mehr erkennen, dass die Instruktion zur Laufzeit anders verwendet wird. Um dies zu erreichen, wird eine spezielle Funktion `trampoline` eingeführt:

```
1 trampoline:
2     pushfd                                ; Flags sichern
3     add dword ptr [esp + 4 * esi], edi    ; Trampolin-Effekt
```

```
4       popfd                                   ; Flags wiederherstellen
5       ret                                     ; Funktion verlassen
```

Isoliert betrachtet ist die Funktion unverdächtig. Als zentrale Operation wird eine über Register konfigurierbare Zelle im Stack verändert, indem der Wert eines anderen Registers addiert wird. Damit die Funktion von beliebiger Stelle aufgerufen werden kann, werden die Flags des Prozessors vor der Addition gesichert und danach wiederhergestellt, da die Addition einen Einfluss auf die Flags hat. Der Trick ist es nun, das Register ESI vor dem Aufruf mit dem Wert 1 zu belegen. Dadurch modifiziert die Addition die Rücksprungadresse der Funktion, die auf dem Stack an Position ESP + 4 gespeichert ist (vgl. Abbildung 5 auf Seite 13). Über das Register EDI kann nun gesteuert werden, wie die Rücksprungadresse modifiziert werden soll. Wird ESI mit dem Wert 2 belegt, so führt die `ret`-Anweisung der Funktion nicht direkt hinter den aufrufenden `call`-Befehl, sondern 2 Bytes dahinter. Dies kann beispielsweise folgendermaßen genutzt werden:

```
1  BE01000000            mov esi, 1
2  BF02000000            mov edi, 2
3                              ...
4  39C8                  cmp eax, ecx
5  E8C2FFFFFF            call trampoline
6  C7860F8D430000009090  mov [esi + 438D0Fh], 9090h
```

Auch dieser Code ist isoliert betrachtet völlig unverdächtig. Zusammen mit der Trampolinfunktion passiert aber zur Laufzeit folgendes: Da ESI auf 1 und EDI auf 2 gesetzt sind, wird nach dem Aufruf der Trampolinfunktion in Zeile 5 nicht der `mov`-Befehl in Zeile 6 ausgeführt, sondern die ersten 2 Bytes davon übersprungen. Die Decodierung dieser Bytefolge ergibt einen anderen Befehl:

```
1  0F8D43000000          jge +49h ; Springt um +49 Bytes
2  90                    nop
3  90                    nop
```

In den Parametern der `mov`-Anweisung war also ein völlig anderer Befehl versteckt, der die bedingte Sprunganweisung zur Vergleichsanweisung in Zeile 4 des vorletzten Listings realisiert. Sowohl die Trampolinfunktion als auch der aufrufende Code sind aus Sicht des Disassemblers und des Angreifers unverdächtige Codeabschnitte. Erst die Kombination beider Teile liefert einen Effekt, der in einer Hochsprache nicht erreicht werden könnte und das Verstecken von Befehlen innerhalb von anderen Befehlen ermöglicht. Im Gegensatz zu Balachandrans Verfahren ist hierzu kein selbstmodifizierender Code nötig, dieser Ansatz funktioniert also auch in nebenläufigen Programmen und hat zudem nicht das Problem, dass eine Modifikation am Sprungziel umgekehrt werden muss. Es eignet sich daher zum Verstecken beliebiger Befehle.

Dieses Verfahren hat allerdings weiterhin den Nachteil, dass die Kenntnis des Verfahrens dem Angreifer eine Umkehr der Transformationen und damit die Aufdeckung von versteckten Befehlen ermöglicht. Er muss dazu lediglich Funktionen identifizieren, die eine parametrisierte Änderung der Rücksprungadresse einer Funktion erlauben und deren Aufrufe untersuchen.

4.2 Ein neues Verfahren

In diesem Abschnitt wird ein neues Verfahren vorgestellt, das dem Angreifer selbst bei Kenntnis des Verfahrens keine Vorteile bei einer statischen Analyse bietet, sofern das Programm nur statisch analysiert wird. Damit ein solches Verfahren tatsächliche Sicherheit bieten kann, werden Informationen als Grundlage für die Verschleierung benötigt, die ein Compiler zwar automatisch aus dem Quelltext berechnen kann, die aber innerhalb einer statischen Analyse nicht ermittelt werden können. Da das Quellprogramm und das Maschinenprogramm aber dieselbe Semantik besitzen sollten, kann dies nicht möglich sein.

Aus dieser Überlegung entstand die Idee, den Programmierer in die Verschleierung in Form eines „kreativen Orakels“ einzubeziehen. Er kann Informationen beisteuern, die *nicht* intuitiv berechenbar und nach der Church-Turing-These dadurch auch nicht von einer Turing-Maschine berechnet werden können.

4.2.1 Unentscheidbare Eigenschaften

Bei einem gegebenen Kontrollflussgraph (nach Definition in Abschnitt 2.3.1 auf Seite 18) eines Programms ist es im Allgemeinen nicht berechenbar, ob ein bestimmter Knoten des Graphen erreicht wird. Das zugehörige Entscheidungsproblem wird in diesem Abschnitt als *Erreichbarkeitsproblem* bezeichnet. Der Grund für dessen Unberechenbarkeit ist, dass es Zyklen im Graphen geben kann, die eine Endlosschleife bilden. Formal lässt sich die Aussage beweisen, indem das Halteproblem auf dieses Problem reduziert wird:

- Angenommen, es gibt einen Löser $\mathcal{L}$ für das Erreichbarkeitsproblem im Kontrollflussgraph. Dieser hat als Eingabe einen Kontrollflussgraph und einen Knoten, von dem geprüft werden soll, ob er erreichbar ist.
- Sei nun p eine Instanz des Halteproblems, codiert als Eingabe für eine universelle Turing-Maschine
- Da die x86-Architektur eine universelle Turingmaschine simulieren kann, sei $\mathcal{S}$ ein solcher Simulator. Dann gibt es mit $\mathcal{S}(p)$ einen Kontrollflussgraph für die Instanz des Halteproblems. Der Knoten, der im Simulator den haltenden Zustand der Turingmaschine simuliert, werde v_h genannt.
- Nun kann der Löser $\mathcal{L}$ für das Erreichbarkeitsproblem das Halteproblem lösen, indem geprüft wird, ob $\mathcal{L}(\mathcal{S}(p), v_h)$ wahr ist, d.h. ob der Knoten v_h im Kontrollflussgraph der simulierten Turingmaschine für die Instanz des Halteproblems erreicht werden kann.
- Da $\mathcal{L}(\mathcal{S}(p), v_h)$ genau dann wahr ist, wenn p eine positive Instanz des Halteproblems ist, wurde das Halteproblem auf das Erreichbarkeitsproblem reduziert, was zu beweisen war. □

Da nicht automatisiert berechnet werden kann, ob ein bestimmter Knoten im Kontrollflussgraph erreichbar ist, kann die Ausführungsreihenfolge der Knoten ebenfalls nicht berechnet werden. Dieses Problem wird als Ausführungsreihenfolgeproblem bezeichnet. Eine Lösung des

Problems wäre durch Prüfung des gesamten Zustandsraumes (analog zum Model-Checking) zwar theoretisch möglich, aber da die Größe des Zustandsraums exponentiell mit der Anzahl der benutzten Variablen wächst, ist dies in der Praxis unpraktikabel. Eine weitere Lösungsidee wäre die Simulation des Programms, allerdings ist diese im Rahmen einer statischen Analyse nicht möglich, da es sich dabei um eine dynamische Analyse handeln würde. Dieses Argument wird durch die Tatsache verstärkt, dass die analysierten Programme sich in Abhängigkeit ihrer Umgebung bei jedem Lauf unterschiedlich verhalten können – es könnte zum Beispiel ein zeitliches Ereignis geben, auf welches das Programm reagiert. In der Simulation könnte dies nicht festgestellt werden, wenn nicht alle Zustände der Umgebung simuliert werden.

Die Idee für das Verfahren ist nun, eine Instanz des Ausführungsreihenfolgeproblems im Programm zu codieren, dessen Lösung dem Compiler bekannt ist. Es gibt also ein Orakel, das dem Compiler eine geordnete Folge von Knoten $v_{o_1}, v_{o_2}, \ldots, v_{o_n}$ im Kontrollflussgraph vorgibt, die folgende Bedingung erfüllt: *Wenn* ein Knoten v_{o_i} mit $i > 1$ zur Laufzeit erreicht wird, dann wurde der Knoten $v_{o_{i-1}}$ vorher bereits genau einmal zur Laufzeit erreicht. Daraus folgt, dass es eine Teilfolge beginnend bei v_{o_1} bis zum Knoten v_{o_j} mit $j > i$ geben kann, sodass die Ausführungsreihenfolge der Knoten genau dieser Ordnung entspricht. Dahinter kann es weitere Knoten in der Folge geben, die zur Laufzeit nicht erreicht werden. Ein Beispiel in der Hochsprache C:

```
1  int decision(int x) {
2    int y = 0;
3    if(x < 10) {
4      y = 23;
5    else if(x > 20) {
6      y = 42;
7    }
8    return y;
9  }
```

Werden die Zeilennummern zur Veranschaulichung als Knoten des Kontrollflussgraphs interpretiert, so ist die Folge 2, 3, 8 eine gültige Folge nach der obigen Bedingung, da diese Zeilen genau einmal in dieser Reihenfolge ausgeführt werden. Die Folge 2, 6, 8 verletzt die Bedingung, da Zeile 6 nur in manchen Läufen erfüllt wird. Ein weiteres Beispiel:

```
1  void loop() {
2    int x = 10;
3    while(1) {
4      x--;
5    }
6    x = 0;
7  }
```

Die Folge 2, 6 ist eine gültige Folge, obwohl Zeile 6 nie erreicht wird – es ist nur gefordert, dass *erreichbare* Knoten zur Laufzeit in der angegebenen Reihenfolge ausgeführt werden.

Steht dem Compiler ein Orakel für solche Folgen zur Verfügung, so kann damit Code generiert werden, mit dem das Programm zur Laufzeit Konstanten berechnen kann, deren Wert im Rahmen einer statischen Analyse nicht berechnet werden kann. Zum Verständnis sind einige Vorbetrachtungen nötig.

$\oplus$	**0**	**1**
0	0	1
1	1	0

Tabelle 16: Die Verknüpfungstafel der xor-Gruppe mit der Länge 1 lässt bereits erkennen, dass es sich um eine abelsche Gruppe handelt

4.2.2 Die xor-Gruppe

Zur Umsetzung des Verfahrens wird eine mathematische Gruppe benötigt, die möglichst effizient auf der x86-Architektur berechnet werden kann. Die `xor`-Anweisung scheint hierfür geeignet zu sein, da die exklusive Oder-Verknüpfungen auf Bitstrings fester Länge eine Gruppe bildet, was nun bewiesen wird. Dazu werden die Gruppenaxiome zuerst auf Bitstrings der Länge 1 bewiesen:

- Abgeschlossenheit: die Elemente der Gruppe bilden die Menge $\{0, 1\}$. Die Rechenvorschrift in der Tabelle 16 zeigt, dass bei allen Verknüpfungen der Gruppenelemente nur Elemente der Gruppe entstehen. Die Gruppe ist also abgeschlossen.
- Assoziativität: aus der Verknüpfungstafel kann ebenfalls abgelesen werden, dass die Verknüpfung assoziativ ist, denn es gelten:

$$
\begin{aligned}
0 \oplus (0 \oplus 0) &= 0 = (0 \oplus 0) \oplus 0 \\
0 \oplus (0 \oplus 1) &= 1 = (0 \oplus 0) \oplus 1 \\
0 \oplus (1 \oplus 0) &= 1 = (0 \oplus 1) \oplus 0 \\
0 \oplus (1 \oplus 1) &= 0 = (0 \oplus 1) \oplus 1 \\
1 \oplus (0 \oplus 0) &= 1 = (1 \oplus 0) \oplus 0 \\
1 \oplus (0 \oplus 1) &= 0 = (1 \oplus 0) \oplus 1 \\
1 \oplus (1 \oplus 0) &= 0 = (1 \oplus 1) \oplus 0 \\
1 \oplus (1 \oplus 1) &= 1 = (1 \oplus 1) \oplus 1
\end{aligned}
$$

  Damit ist die Assoziativität der Verknüpfung gezeigt.

- Neutrales Element: Aus der Verknüpfungstafel ist ersichtlich, dass 0 das neutrale Element der Verknüpfung ist, denn es gelten:

$$
\begin{aligned}
0 \oplus 0 &= 0 \\
1 \oplus 0 &= 1 \\
0 \oplus 1 &= 1
\end{aligned}
$$

  Damit ist gezeigt, dass 0 linksneutral und rechtsneutral, also neutral ist.

- Inverses Element: Aus der Verknüpfungstafel ist ersichtlich, dass jedes Element zu sich selbst invers ist, denn es gelten:

$$0 \oplus 0 = 0$$
$$1 \oplus 1 = 0$$

Insgesamt folgt, dass die xor-Verknüpfung auf Bitstrings der Länge 1 eine Gruppe ist. Aus der Verknüpfungstafel ist ersichtlich, dass es sich sogar um eine abelsche Gruppe handelt, die xor-Verknüpfung also kommutativ ist. Der Beweis, dass dies für beliebige Länge gilt, wird durch vollständige Induktion erbracht:

- Induktionsbehauptung: Die Verknüpfung $(\{0,1\}^n, \oplus)$ mit elementweiser xor-Verknüpfung ist für beliebige $n \in \mathbb{N}$ eine Gruppe.
- Induktionsanfang: Oben wurde bereits gezeigt, dass die Behauptung für $n = 1$ gilt.
- Induktionsvoraussetzung: Die Verknüpfung $(\{0,1\}^n, \oplus)$ sei für beliebiges, aber festes $n \in \mathbb{N}$ eine Gruppe.
- Induktionsschritt: Da die Verknüpfung elementweise erfolgt, beeinflussen die Elemente des Bitstrings sich nicht gegenseitig. Die Verknüpfung erfolgt folgendermaßen:

$$(x_1, x_2, \ldots, x_n) \oplus (y_1, y_2, \ldots y_n) = (x_1 \oplus y_1, x_2 \oplus y_2, \ldots x_n \oplus y_n)$$

  Für den Übergang zu $n + 1$ gibt es folgende 4 Fälle:

$$(x_1, x_2, \ldots, x_n, 0) \oplus (y_1, y_2, \ldots y_n, 0) = (x_1 \oplus y_1, x_2 \oplus y_2, \ldots x_n \oplus y_n, 0)$$
$$(x_1, x_2, \ldots, x_n, 0) \oplus (y_1, y_2, \ldots y_n, 1) = (x_1 \oplus y_1, x_2 \oplus y_2, \ldots x_n \oplus y_n, 1)$$
$$(x_1, x_2, \ldots, x_n, 1) \oplus (y_1, y_2, \ldots y_n, 0) = (x_1 \oplus y_1, x_2 \oplus y_2, \ldots x_n \oplus y_n, 1)$$
$$(x_1, x_2, \ldots, x_n, 1) \oplus (y_1, y_2, \ldots y_n, 1) = (x_1 \oplus y_1, x_2 \oplus y_2, \ldots x_n \oplus y_n, 0)$$

  Das letzte Bit wird also analog zur Gruppe der Länge 1 erzeugt, wobei die vorigen Bits nach Induktionsvoraussetzung bereits eine Gruppe bilden. Daher handelt es sich auch bei der xor-Verknüpfung von Bitstrings der Länge $n + 1$ um eine Gruppe, was zu beweisen war. □

Für die Umsetzung des Verfahrens werden Bitstrings der Länge 32 genutzt, denn dies entspricht genau der Wortbreite der x86-Architektur. Die Gruppeneigenschaft ist wichtig, da Gruppen eine für das Verfahren wichtige Eigenschaft besitzen: In Gruppen sind alle Gleichungen eindeutig lösbar, d.h. es gilt

$$\forall x, y \in \{0,1\}^{32}\ \exists! a \in \{0,1\}^{32} : x \oplus a = y$$

Jeder Bitstring kann also durch xor-Verknüpfung mit einem eindeutigen anderen Bitstring in einen beliebigen Bitstring überführt werden. Soll a durch Verknüpfung in b überführt werden, so ist dies durch Verknüpfung von a mit dem Wert von $a \oplus b$ möglich: $a \oplus (a \oplus b) = (a \oplus a) \oplus b = 0 \oplus b = b$. Die Gruppe $(\{0,1\}^{32}, \oplus)$ wird im Folgenden als G bezeichnet.

4.2.3 Die rnd-Funktion

Weiterhin wird eine Familie von Zufallsfunktion $rnd_i : \mathbb{N} \to \{0,1\}^{32}$ mit $i \in \mathbb{N}$ benötigt. Die Funktion bildet natürliche Zahlen auf einen zufälligen Bitstring ab. Die Zuordnung von Zahl zu Bitstring soll dabei fest sein, sodass $rnd_i(j)$ für feste i, j immer denselben Wert liefert. Der Funktionswert soll in dem Sinne zufällig sein, dass es für einen Angreifer nicht möglich ist, aus einer Folge von Funktionswerten $rnd_i(j_1), rnd_i(j_2), \ldots, rnd_i(j_n)$ mit festem i eine Ordnung bezüglich der Argumente $j_1, j_2, \ldots, j_n$ zu bestimmen. Aus gegebenen Funktionswerten $rnd_i(j)$ und $rnd_i(k)$ darf also nicht berechenbar sein, ob $j < k$ gilt. Die Funktion wird nur innerhalb des Compilers benötigt. Die Implementierung kann deshalb so erfolgen, dass die Funktion in einer Datenstruktur speichert, welche Argumente j für ein gegebenes i bereits berechnet wurden: Wenn der Wert (i, j) erstmalig benötigt wird, so wird der Funktionswert zufällig erzeugt (z.B. durch die Nutzung der Entropiequelle des Betriebssystems) und in der Datenstruktur für das Tupel (i, j) abgelegt. Bei erneuter Abfrage kann der Funktionswert dann aus der Datenstruktur gelesen werden.

4.2.4 Erzeugung verschleierter Konstanten

Mit diesen Hilfsmitteln kann der Compiler nun Code erzeugen, mit dem das Programm zur Laufzeit Konstanten berechnen kann, die in einer rein statischen Analyse nicht berechnet werden können. Als Orakel für das Ausführungsreihenfolgeproblem dienen Annotationen des Programmierers. Im Quelltext der Hochsprache können beliebige Anweisungen im Programm mit einer natürlichen Zahl annotiert werden, um den Compiler darauf hinzuweisen, dass die geordnete Folge dieser Annotationen die oben formulierte Bedingung der Folge einhält. Zur Berechnung beliebiger Konstanten wird vom Compiler eine einzelne, globale Variable s in das Programm eingeführt, aus der sich später beliebig viele Konstanten ableiten lassen. Der Wert der Variable s wird mit $rnd_0(0)$ initialisiert. Nun kann an jeder mit $i > 0$ annotierten Stelle des Programms folgendes Verfahren durch den Compiler angewendet werden:

- wenn die Stelle mit $i \in \mathbb{N}_{>0}$ annotiert ist, berechne $r_i = rnd_0(i)$
- erzeuge vor der annotierten Anweisung folgende Anweisung im Maschinencode:

$$s \leftarrow s \oplus r_i$$

Die Bedingung an die annotierte Folge von Programmpunkten garantiert, dass s zur Laufzeit nacheinander die Werte der Folge mit den Elementen

$$s_i = \bigoplus_{j=0}^{i} r_j$$

annimmt, wobei i für den zuletzt erreichten Programmpunkt steht. Bei der Initialisierung hat s den Wert $s_0 = r_0$, nach dem ersten Programmpunkt den Wert $s_1 = r_0 \oplus r_1$, nach dem zweiten Programmpunkt den Wert $s_2 = r_0 \oplus r_1 \oplus r_2$ und so weiter bis zum Wert s_n, wenn n der Index des letzten Programmpunkts ist, der zur Laufzeit erreicht wird. Es können weitere Programmpunkte mit Werten größer n annotiert werden, die zur Laufzeit nicht erreicht werden, weil sie beispielsweise hinter einer Endlosschleife verborgen sind. Dies ist in Programmen mit grafischer Oberfläche einfach erreichbar, da solche Programme üblicherweise eine Hauptschleife zur Abarbeitung von Ereignissen besitzen und diese nie verlassen wird, während das Programm abläuft. Wird der letzte Programmpunkt hinter dieser Schleife gesetzt, ist die Bedingung an die Folge trotzdem erfüllt.

Soll nun eine Konstante verschleiert werden, so kann dies im Quelltext entsprechend annotiert werden. Dazu muss neben dem Wert der zu verschleiernden Konstante v auch angegeben werden, welcher Programmpunktindex i_v beim Erreichen der Stelle des Zugriffs auf die Konstante zuletzt zur Laufzeit erreicht wurde. Der Programmierer muss also zusichern, dass beim Zugriff auf die Konstante v zuletzt der mit der Zahl i_v annotierte Programmpunkt erreicht wurde. Statt der Konstante v kann der Compiler nun den Wert $\hat{v} := v \oplus s_{i_v} = v \oplus r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v}$ generieren. Die Werte der Zufallszahlen $r_0, \ldots, r_{i_v}$ zur Berechnung von $\hat{v}$ sind durch den Compiler berechenbar, selbst wenn der Programmpunkt im Quelltext erst hinter dem Zugriff auf die Konstante liegt, denn es gilt $r_i = rnd_0(i)$. Damit zur Laufzeit der korrekte Wert v berechnet wird, generiert der Compiler hinter dem Laden der verschleierten Konstante $\hat{v}$ eine exklusive Oder-Verknüpfung mit der globalen Variable s. Die Bedingung an die Annotierung der Programmpunkte garantiert, dass s zur Laufzeit an dieser Stelle genau den Wert $s_{i_v} = r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v}$ hat.

Zur Laufzeit wird der vom Compiler erzeugte, verschleierte Wert $\hat{v}$ also mit dem Wert an der Speicherstelle der Variable s verknüpft, die bei korrekter Annotation durch den Programmierer zu diesem Zeitpunkt genau den Wert s_{i_v} enthält. Durch die Verknüpfung entsteht zur Laufzeit also genau der Wert v, denn es gilt:

$$\begin{aligned}
\hat{v} \oplus s =& (v \oplus s_{i_v}) \oplus s \\
=& (v \oplus r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v}) \oplus s \\
=& (v \oplus r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v}) \oplus (r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v}) \\
=& v \oplus ((r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v}) \oplus (r_0 \oplus r_1 \oplus \ldots \oplus r_{i_v})) \\
=& v \oplus 0 \\
=& v
\end{aligned}$$

i	$\mathbf{rnd_0(i)}$
0	0xAC2A
1	0x9AF3
2	0xB56E
3	0x709A
4	0x6382

Tabelle 17: Der interne Zustand der *rnd*-Funktion für das Beispiel. Die Werte werden beim ersten Zugriff auf $rnd(i)$ zufällig bestimmt und gespeichert, um bei weiteren Aufrufen von $rnd(i)$ denselben Wert zurückgeben zu können.

Dadurch wird zur Laufzeit der korrekte Wert v aus dem verschleierten Wert berechnet. Der Angreifer kennt den aktuellen Wert der Variable s aber nicht und kann diese Berechnung daher nicht durchführen. Eine Analyse der Schreibzugriffe auf s wird ihm jeden Programmpunkt zeigen – ohne die Kenntnis der Reihenfolge der Zugriffe wird er aber trotzdem nicht berechnen können, welchen Wert s an der Stelle der Entschleierungsberechnung besitzt. Der folgende Ausschnitt zeigt ein Beispiel für solche Annotationen:

```
1  int checkPassword(int given) {
2    // konstante schuetzen, letzter programmpunkt: 3
3    return given == 0x1234;
4  }
5
6  int main() {
7    // programmpunkt 1
8    initGUI();
9    while(1) {
10     handleGUIEvents(); // ruft ggf. checkPassword auf
11   }
12   // programmpunkt 4
13   return 0;
14 }
15
16 void initGUI() {
17   // programmpunkt 2
18   ...
19   // programmpunkt 3
20 }
```

Der Code beginnt bei `main`, also wird zur Laufzeit zuerst Programmpunkt 1 und danach die Programmpunkte 2 und 3 innerhalb von `initGUI` erreicht. Programmpunkt 4 wird nicht erreicht, da es sich bei dem Programm um eine grafische Anwendung handelt, die vom Betriebssystem auf andere Weise terminiert wird, sodass `main` als Endlosschleife abläuft. Die Funktion `checkPassword` benutzt eine Konstante, die verschleiert werden soll. Dazu ist im Quellcode annotiert, dass der Programmpunkt 3 beim Zugriff auf die Konstante zuletzt erreicht wurde. Dies ist korrekt, da die Funktion aus der GUI heraus aufgerufen wird, sodass Programmpunkt 3 vorher erreicht wurde. Programmpunkt 4 kann nicht erreicht worden sein, da es sich um eine Endlosschleife handelt.

Die Reihenfolge beim Übersetzen ist eine andere als zur Laufzeit, der Compiler übersetzt zuerst die Funktion `checkPassword`. Da die Konstante 0x1234 geschützt werden soll und Programmpunkt 3 laut Annotation an dieser Stelle zuletzt erreicht wurde, benötigt der Compiler die Werte r_0, r_1, r_2 und r_3. Diese seien o.B.d.A. die in Tabelle 17 aufgeführten Werte der *rnd*-Funktion (zur Vereinfachung mit 16 statt 32 Bit). Ohne Verschleierung würde der Compiler für Zeile 3 etwa folgenden Code generieren (Parameter im Register EBX, Rückgabewert im Register EAX):

```
1  mov eax, 1234h ; Laden der Konstante
2  cmp ebx, eax   ; Vergleich mit dem Parameter
3  setz al        ; Speichern des Vergleichsergebnisses im Rueckgaberegister
```

Bei aktivierter Verschleierung wird dem Compiler garantiert, dass der Programmpunkt 3 zuletzt erreicht wurde. Der Wert der globalen Variable s ist zur Laufzeit an dieser Stelle also

$$\begin{aligned} s_3 &= r_0 \oplus r_1 \oplus r_2 \oplus r_3 \\ &= rnd_0(0) \oplus rnd_0(1) \oplus rnd_0(2) \oplus rnd_0(3) \\ &= \text{0xAC2A} \oplus \text{0x9AF3} \oplus \text{0xB56E} \oplus \text{0x709A} \\ &= \text{0xF32D} \end{aligned}$$

Da der Compiler den Laufzeitwert von s auf diese Weise berechnen kann, kann die Konstante nun mit dem Wert von s_3 verschleiert werden. Statt der Konstante $v = 0x1234$ wird der Wert $\hat{v} = \text{0x1234} \oplus \text{0xF32D} = \text{0xE119}$ im Programmcode generiert. Die Entschleierung der Konstante findet zur Laufzeit durch Verknüpfung mit der globalen Variable s statt, sodass der verschleierte Code nun folgende Gestalt annimmt:

```
1  mov eax, 0E119h         ; Laden der verschleierten Konstante
2  xor eax, dword ptr [s]  ; Entschleiern mit dem Wert von s
3  cmp ebx, eax            ; Vergleich mit dem Parameter
4  setz al                 ; Speichern des Vergleichsergebnisses
```

Der Programmcode ist also nur um eine einzelne Anweisung angewachsen, nämlich die zur Entschleierung der Konstante. Auch an den annotierten Programmpunkten selbst wird nur eine einzelne Anweisung generiert, nämlich die Anweisung zur Aktualisierung von s. An Programmpunkt 1 sieht diese folgendermaßen aus:

```
1  xor dword ptr [s], 9AF3h ; Aktualisierung von s mit r_1
```

An den anderen Programmpunkten i analog mit den Werten von $rnd_0(i)$. Der Startwert für s ist im Datensegment als $rnd_0(0) = \text{0xAC2A}$ abgelegt. Das Verfahren kann also benutzt werden, um beliebige 32-Bit-Konstanten zu verschleiern und zur Laufzeit mit einer einzigen Anweisung wieder zu entschleiern.

4.2.5 Analyse und Verbesserung

Um das Verfahren über eine statische Analyse anzugreifen, müsste der Angreifer herausfinden, welche Programmpunkte beim Zugriff auf eine Konstante bereits erreicht wurden. Dann könnte er den Laufzeitwert von s rekonstruieren und die Konstante entschleiern. Dieser Angriff kann aber nicht automatisiert werden, da diese Eigenschaft des Programms nicht berechenbar ist. Der Angreifer müsste also seine Kreativität nutzen und das Programm manuell angreifen. Aus dem gleichen Grund müssen die Annotationen manuell durch den Programmierer durchgeführt werden – die Spezialfälle, in denen der Compiler die Reihenfolge selbst berechnen könnte, wären ebenfalls in einer statischen Analyse berechenbar.

Eine andere Angriffsmöglichkeit ist, die Menge *aller* möglichen Laufzeitwerte von s zu berechnen und diese mit der verschleierten Konstante zu verknüpfen, um daraus die Menge aller möglichen entschleierten Werte zu erhalten. Im obigen Beispiel kann s unter anderem die folgenden Werte annehmen:

$$
\begin{aligned}
r_0 &= \text{0xAC2A} && = \text{0xAC2A}\\
r_0 \oplus r_1 &= \text{0xAC2A} \oplus \text{0x9AF3} && = \text{0x36D9}\\
r_0 \oplus r_1 \oplus r_2 &= \text{0xAC2A} \oplus \text{0x9AF3} \oplus \text{0xB56E} && = \text{0x83B7}\\
r_0 \oplus r_1 \oplus r_2 \oplus r_3 &= \text{0xAC2A} \oplus \text{0x9AF3} \oplus \text{0xB56E} \oplus \text{0x709A} && = \text{0xF32D}\\
r_0 \oplus r_1 \oplus r_2 \oplus r_3 \oplus r_4 &= \text{0xAC2A} \oplus \text{0x9AF3} \oplus \text{0xB56E} \oplus \text{0x709A} \oplus \text{0x6382} && = \text{0x90AF}\\
&\ldots
\end{aligned}
$$

Die möglichen Werte für die verschleierte Konstante 0xE119 sind somit unter anderem 0x4D33, 0xD7C0, 0x62AE, 0x1234 und 0x71B6. Es gibt noch weitere Möglichkeiten, denn da der Angreifer die Reihenfolge der Programmpunkte nicht kennt, wäre $r_0 \oplus r_3$ ebenfalls ein möglicher Wert für s. Dieser Angriff könnte dem Angreifer möglicherweise helfen, wenn die korrekte Konstante danach einfach ausgewählt werden kann. Ein Brute-Force-Angriff dieser Art soll daher erschwert werden. Im bisher beschriebenen Verfahren gibt es bei der Verwendung von n Programmpunkten genau $2^n + 1$ mögliche Werte für die Variable s, nämlich den Startwert sowie die Anzahl aller möglichen Teilmengen der Menge $\{r_1, \ldots, r_n\}$. Die Reihenfolge der Programmpunkte ist bei diesem Angriff nicht relevant, da die xor-Verknüpfung kommutativ ist. Die Kommutativität der Verknüpfung wird nun beseitigt, indem nach jeder Aktualisierung von s die Bits um einen konstanten Wert nach links gerollt werden. Diese Operation wurde gewählt, da hierdurch die Entropie von s nicht sinkt und es für diese Operation eine Anweisung in der x86-Architektur gibt, nämlich den Befehl `rol`. Die Hintereinanderausführung von xor-Verknüpfung und Bit-Rollen werde mit dem Operator $\oplus_<$ bezeichnet. Alle obigen Betrachtungen gelten weiterhin, allein die Kommutativität der Operation geht verloren. Es wird nun betrachtet, wie viele Möglichkeiten es bei n Programmpunkten für den Laufzeitwert von s gibt. Dies entspricht genau der Anzahl *aller* möglichen Folgen, die aus einer beliebigen Auswahl von Werten aus der Menge $\{r_1, \ldots, r_n\}$ in gebildet werden können. Bei n Werten kann die Folge Längen von 0 bis n annehmen, da auch

die leere Folge erlaubt ist. Für jede Folge der Länge k gibt es $\binom{n}{k}$ Möglichkeiten, Elemente aus der Menge zu wählen. Da die Reihenfolge der Elemente in der Folge eine Rolle spielt, erhöht sich die Anzahl der Teilfolgen der Länge k auf $\binom{n}{k} \cdot k!$. Insgesamt gilt für die Anzahl aller Teilfolgen also:

$$
\begin{aligned}
&\sum_{k=0}^{n} \binom{n}{k} k! \\
=&\sum_{k=0}^{n} \frac{n!}{k! \cdot (n-k)!} \cdot k! \\
=&\sum_{k=0}^{n} \frac{n!}{(n-k)!} \\
=&n! \cdot \sum_{k=0}^{n} \frac{1}{(n-k)!} \\
=&n! \cdot \left(\frac{1}{(n-0)!} + \frac{1}{(n-1)!} \ldots + \frac{1}{(n-n)!} \right) \\
=&n! \cdot \left(\frac{1}{(n-n)!} + \ldots + \frac{1}{(n-1)!} + \frac{1}{(n-0)!} \right) \\
=&n! \cdot \left(\frac{1}{0!} + \ldots + \frac{1}{(n-1)!} + \frac{1}{n!} \right) \\
=&n! \cdot \sum_{k=0}^{n} \frac{1}{k!}
\end{aligned}
$$

Die verbleibende Summe konvergiert schnell gegen die Eulersche Zahl e (sie ist eine der Definitionen dieser Zahl). Die Anzahl der Folgen kann somit als $e \cdot n!$ abgeschätzt werden. Der genaue Wert ist $\lfloor e \cdot n! \rfloor$ (nach [IS14]). Bereits für 12 Programmpunkte überschreitet die Anzahl der möglichen Anordnungen in einem Brute-Force-Angriff dabei den Wert 2^{32} – der Angreifer hat durch die Kenntnis der Zufallswerte der Programmpunkte also keinen Vorteil mehr, weil bei einem Brute-Force-Angriff *alle* Werte möglich sein könnten, die überhaupt im 32-Bit-Register berechnet werden können.

Es handelt sich beim vorgestellten Verfahren um einen neuen Ansatz für die Verschleierung von Programmen. Dazu wird die menschliche Kreativität des Programmierers genutzt, um die Instanz eines im Rahmen einer statischen Analyse nicht berechenbaren Problems im Programm zu codieren, dessen Lösung das Programm selbst durch seine eigene Ausführung berechnet. Zwar wäre es möglich, das Verfahren durch Simulation des Programms oder durch eine dynamische Analyse einfach anzugreifen – dazu müsste lediglich ein Lauf des Programms aufgezeichnet werden, bei dem alle relevanten Programmpunkte durchlaufen werden. Allerdings kann ein solcher Lauf nicht immer erhalten werden – wenn das Programm auf zeitliche Ereignisse, bestimmte Netzwerkpakete oder andere Ereignisse der äußeren Umgebung reagiert und einige Programmpunkte davon abhängen, kann ein vollständiger Lauf durch die Programmpunkte auch nur in der korrekten Umgebung stattfinden. Im Vergleich zu

den anderen vorgestellten Verfahren ist dies das einzige, das eine rein statische Analyse selbst bei Kenntnis des Verfahrens verhindern kann. Im Vergleich zum vorgestellten Verfahren für opake Konstanten ist es dazu schneller durchzuführen, da nur 2 Befehle pro Programmpunkt und Entschleierung benötigt werden. Der benötigte Speicherplatz ist dabei gering, es wird nur Platz für eine einzige globale Variable benötigt.

4.3 Verschleierung von Zeichenketten

Zuletzt soll noch ein einfaches Verfahren in den Compiler integriert werden, das zwar sehr einfach angreifbar ist, aber wegen der großen Heimlichkeit bei spärlicher Anwendung trotzdem sinnvoll eingesetzt werden kann. In Abschnitt 2.5 auf Seite 30 wurde beschrieben, dass der Angreifer die für ihn relevanten Stellen im Maschinencode oft dadurch findet, dass in der Black-Box-Analyse auftauchende Zeichenketten gesucht und die referenzierenden Stellen im Maschinencode untersucht werden. Die bisher gezeigten Verfahren ermöglichen es, den Zugriff auf die Zeichenketten zu verschleiern, sodass der Angreifer zwar die eigentliche Zeichenkette im Programm finden kann, nicht aber die Stellen im Maschinencode, die auf diese zugreifen. Durch die Integration von Obfuscation in einen Compiler ist es möglich, die Zeichenketten selbst zu verschleiern, sodass diese im Datensegment des Programms nicht mehr auftauchen. Um die Semantik des Programms nicht zu verletzen, muss dabei entweder sichergestellt werden, dass auch schreibende Zugriffe auf die Zeichenkette die Verschleierung berücksichtigen oder dass nur Zeichenketten verschleiert werden, die als Literal in der Hochsprache auftauchen, also konstant sind. Diese Unterscheidung kann auf Assembler-Ebene nicht mehr nachvollzogen werden, da die Datentypen hier nicht mehr auftauchen. Der Compiler kann während der Übersetzung allerdings unterscheiden, ob eine Zeichenkette als Literal benutzt wird oder ein beschreibbares Array von Zeichen darstellt, das mit einem bestimmten Wert initialisiert ist. Im nun vorgestellten Ansatz werden nur Literale verschleiert.

Die Idee dazu ist, eine im Quelltext annotierte Zeichenkette v durch eine *andere*, ebenfalls in der Annotation angegebene Zeichenkette u zu ersetzen. Die verdächtige Zeichenkette v soll im Datensegment der Anwendung nicht auftauchen – stattdessen soll die unverdächtige Zeichenkette u im Datensegment gespeichert werden. Erst unmittelbar vor dem Zugriff auf v soll die unverdächtige Zeichenkette u zur Laufzeit nach v transformiert werden. Beim Verlassen der benutzenden Funktion soll die transformierte, unverschleierte Zeichenkette v nicht mehr im Speicher verbleiben, damit eine Abbildung des Anwendungsspeichers nicht nach v durchsucht werden kann. Ein Beispiel für eine Verwendung des Verfahrens könnte folgendermaßen aussehen:

```
1  int checkPassword(int given) {
2      if(given != 1234) {
3          // verschleiere als "Help"
4          showError("Invalid Password");
5          return 0;
6      } else {
7          return 1;
8      }
9  }
```

I	n	v	a	l	i	d		S	e	r	i	a	l
49	6E	76	61	6C	69	64	20	53	65	72	69	61	6C
64	65	62	75	67	3A	20	61	6C	6C	20	6F	6B	00
d	e	b	u	g	:		a	l	l		o	k	
2D	0B	14	14	0B	53	44	41	3F	09	52	06	0A	6C

Abbildung 18: Zur Verschleierung einer verdächtigen Zeichenkette v (hier: „Invalid Serial“) wird eine unverdächtige Zeichenkette u (hier: „debug: all ok“) benutzt. Durch die byteweise Berechnung von $k = v \oplus u$ entsteht eine Bytefolge k (unten), die zur Überführung von u nach v benutzt werden kann, denn es gilt $u \oplus k = v$. Im Maschinenprogramm müssen also nur die unverdächtige Zeichenkette u sowie die Bytefolge k gespeichert werden. Erst zur Laufzeit wird daraus v berechnet.

Die tatsächlich verwendete, aus Sicht des Angreifers verdächtige Zeichenkette „Invalid Password“ soll in der ausführbaren Anwendung nicht auftauchen. Stattdessen soll die unverdächtige Zeichenkette „Help“ im Datensegment gespeichert und an der annotierten Stelle referenziert werden. Der Angreifer wird auf diese Weise in die Irre geführt, da die abgelegte Zeichenkette unverdächtig erscheint und der zugehörige Code im besten Fall nicht weiter betrachtet wird. Wenn diese Technik nur für wenige Zeichenketten angewendet wird, fällt dies dem Angreifer weniger auf, als wenn alle Zeichenketten des Programms verschleiert wären, wie es bei bisherigen Obfuscation-Ansetzen der Fall wäre.

4.3.1 Verfahren

Die Umsetzung bedient sich wieder der xor-Gruppe (vgl. Abbildung 18):

- wenn eine Zeichenkette v durch eine Zeichenkette u verschleiert werden soll, bringe zunächst die Längen beider Zeichenketten durch das Anfügen von Zufallsbytes hinter der Null-Terminierung auf das nächste gemeinsame Vielfache von 4. Die auf diese Weise modifizierten Zeichenketten werden $\hat{v}$ und $\hat{u}$ genannt.
- da nun $|\hat{v}| = |\hat{u}| =: l$ gilt, können beide Zeichenketten als Elemente der Gruppe $(\{0,1\}^{8l}, \oplus)$ interpretiert werden, da jedes Zeichen aus 8 Bit besteht. Berechne nun $k = \hat{v} \oplus \hat{u}$.
- lege die Konkatenation $\hat{u}|k$ im Datensegment der Anwendung ab. Auf diese Weise wird auch die unverdächtige Zeichenkette u in der ausführbaren Anwendung abgelegt, da u ein Präfix von $\hat{u}$ ist. Die Länge des abgelegten Werts beträgt genau $2l$, da auch $|k| = \hat{u}$ gilt. Weiterhin gilt, dass $2l$ ein Vielfaches von 4 ist, da die Längen vorher auf das nächste Vielfache von 4 gebracht wurden.

An der Stelle im Hochsprachencode, an der auf das Literal v zugegriffen wird, generiert der Compiler folgenden Code:

- reserviere l Bytes im Prozedurrahmen, die Adresse sei a

- lege die Adresse von $\hat{u}|k$ in ein freies Register r_u
- führe $\frac{l}{4}$ Iterationen durch, Zähler i:
  - berechne $v' \leftarrow [r_u + 4 \cdot i] \oplus [r_u + l + 4 \cdot i]$, an der Stelle $r_u + l$ beginnt k
  - lege v' in $[a + 4 \cdot i]$ ab
- benutze a zum Zugriff auf v

Es werden also jeweils 4 Bytes von $\hat{u}$ und k gleichzeitig xor-verknüpft, um die Register maximal auszunutzen. Auf diese Weise entsteht wegen $\hat{u} \oplus k = \hat{v}$ der Wert der unverschleierten Zeichenkette v im Prozedurrahmen. Da sich die eingefügten Zufallsbytes hinter der Null-Terminierung befinden, werden diese aus Sicht der Zeichenkette ignoriert. Da der Prozedurrahmen beim Verlassen der Prozedur aufgeräumt wird, verbleibt v nicht im Speicher der Anwendung. Durch die Speicherung als Konkatenation kann ein direkter Zugriff auf die Adresse von k vermieden werden, im Maschinencode wird nur die Adresse geladen, an der $\hat{u}$ und somit auch u auftaucht. So wird ein Bezug zur unverdächtigen Zeichenkette im Maschinencode hergestellt, ohne dass das Vorhandensein des Schlüssels k offensichtlich ist.

4.3.2 Analyse

Wenn das Angreifer für alle Zeichenketten der Anwendung angewendet würde, so kann der Angreifer dies schnell bemerken, einen beliebigen Zugriff auf eine solche Zeichenkette analysieren und ein Skript entwerfen, das automatisiert alle Zeichenketten entschleiert. Die Widerstandsfähigkeit des Verfahrens ist also sehr gering. Durch die Integration in einen Compiler steigt die Heimlichkeit allerdings sehr stark – wenn es nur wenige Male angewendet wird, sind alle anderen Zeichenketten noch unverschleiert im Datensegment abgelegt. Dadurch wird es dem Angreifer erschwert, das Vorhandensein der Verschleierung überhaupt zu bemerken. Durch die Tarnung einer Zeichenkette durch eine andere wird zusätzlich für eine Erschwerung gesorgt, da von der entschleiernden Stelle im Maschinencode abgelenkt wird. Dieses Verfahren soll aufzeigen, dass die Integration von Obfuscation in einen Compiler auch Transformationen mit geringer Widerstandsfähigkeit durch Erhöhung der Heimlichkeit verbessern kann.

5 Implementierung durch Integration in einen Übersetzer

Im vorigen Abschnitt wurde erläutert, dass die Integration von Obfuscation in einen Compiler bestehende Ansätze zur Verschleierung verbessern kann und zudem neue Ansätze ermöglicht. Zur praktischen Überprüfung dieser Überlegungen wurden einige der vorgestellten Verschleierungstechniken implementiert.

5.1 Ansatz

Um nach der Implementierung neuer Verschleierungstechniken den Einfluss auf die Laufzeitgeschwindigkeit und die Größe von übersetzten Programmen mit und ohne Verschleierung vergleichen zu können, sollte es für den Benutzer des verschleierungsfähigen Compilers möglich sein, die Verschleierungen selektiv aktivieren oder deaktivieren zu können. Da ein Kriterium für Verschleierungstechniken die Erhaltung der Semantik des verschleierten Programms ist, wäre es zudem wünschenswert, dass der Compiler in der Lage ist, bestehende und weit verbreitete Programme oder Programmbibliotheken zu übersetzen, um auf diese Weise eine Basis für Testfälle zu erhalten. Der Quellcode vieler Programmbibliotheken enthält üblicherweise viele Testfälle, mit denen das korrekte Verhalten der übersetzten Bibliothek getestet werden kann. Diese Testfälle können genutzt werden, um Einflüsse von Verschleierungstechniken auf die Semantik des Programms aufzudecken. Zur Implementierung der Verfahren wurden daher folgende Varianten diskutiert:

- Implementierung eines neuen Compilers: Um Techniken zur Verschleierung auf möglichst vielen Ebenen eines Compilers zu integrieren, wurde die Implementierung eines neuen, speziell zur Erzeugung verschleierten Codes vorgesehenen Compilers erwogen. Hierbei könnten Schnittstellen zur Erweiterung des Compilers durch verschiedene Verschleierungstechniken bereits im Entwurf des Compilers vorgesehen werden, sodass eine experimentelle Umgebung für Obfuscation entstünde. Nachteilig ist allerdings der hohe Aufwand – die Implementierung eines Compilers für eine Hochsprache ist für sich betrachtet schon sehr aufwändig. Es müsste also entweder eine Hochsprache mit einer simplen Grammatik als Quellsprache gewählt werden oder die Grammatik einer weit verbreiteten Hochsprache könnte nur mit Einschränkungen implementiert werden. Bei beiden Varianten wäre es dann allerdings schwierig, Quellcodes von bestehenden Programmen zu Vergleichszwecken zu übersetzen.
- Modifikation eines bestehenden Compilers: Es gibt viele Compiler, deren Quellcode unter einer freien Lizenz zur Verfügung steht. Ein solcher Compiler könnte angepasst werden, um Verschleierungstechniken direkt zu integrieren. Der Vorteil wäre, dass es dann viele Projekte gäbe, die zu Benchmarking-Zwecken mit und ohne Verschleierung übersetzt und verglichen werden könnten. Nachteilig ist allerdings ein großer Aufwand zur Einarbeitung in bestehende Compiler, da diese oftmals aus Millionen Zeilen Quelltext bestehen.

Um die Übersetzung von bestehenden Programmbibliotheken zu ermöglichen und daraus Benchmarks erzeugen zu können, wurde die zweite Variante gewählt. Als Quellsprache wurde

die Programmiersprache C ausgewählt, da viele Programmbibliotheken in C geschrieben sind und andere Programmiersprachen oft nur über Wrapper-Code auf dem C-Code aufbauen. Weiterhin sind Übersetzer für die Programmiersprache C dank der vergleichsweise einfachen Syntax weniger komplex, als es beispielsweise bei objektorientierten Sprachen der Fall wäre.

Die meisten Übersetzer für C erzeugen optimierten Code und erfordern daher viele dynamische Datenstrukturen, um den Code zu verschiedenen Zeitpunkten während des Übersetzens zu optimieren. Dadurch steigt die Komplexität des Compilers, sodass die Einarbeitung in den Code von optimierenden Compilern sich schwierig gestaltet. Der *Tiny C Compiler* (TCC) [Bel13] verfolgt ein anderes Ziel: es wird kein optimierter Code erzeugt, sondern der Compiler selbst ist auf Geschwindigkeit optimiert. Dies hat für dieses Projekt den Vorteil, dass der Compiler mit 30.000 Zeilen C-Code recht überschaubar ist, sodass eine schnelle Einarbeitung erfolgen kann. Trotzdem unterstützt TCC den gesamten Umfang der Sprache C und einiger Erweiterungen des gcc-Projekts, sodass er zur Übersetzung vieler C-Programme und -Bibliotheken genutzt werden kann. In der Praxis wird TCC vorwiegend genutzt, um bei der verteilten Entwicklung von C-Programmen mit Versionsverwaltungssystemen wie SVN oder git automatisiert zu überprüfen, ob eine Änderung des Codes zu einem Übersetzungsfehler führt. Wegen der hohen Geschwindigkeit des Compilers kann diese Prüfung auch bei großen Projekten bei jeder Änderung des Quellcodes erfolgen und die Änderung ggf. abgelehnt werden. Die vollständige Übersetzung des Linux-Kernels benötigt mit TCC nur wenige Sekunden, während optimierende Compiler wie gcc mehrere Minuten benötigen. Die Geschwindigkeit des Übersetzers ist für diese Arbeit allerdings nicht relevant, aber TCC ist unter den freien C-Compilern derjenige mit den wenigsten Zeilen Code und ermöglicht daher eine rasche Umsetzung von Anpassungen.

5.2 Der Tiny C Compiler

TCC weist aufgrund seines Ziels, ein möglichst schneller Compiler zu sein, einige Besonderheiten gegenüber anderen, optimierenden Compilern auf:

- Autarkie: Um möglichst schnell zu sein, erzeugt TCC keinen Assembler-Code, sondern emittiert direkt Maschinencode für die Zielarchitektur. Es ist daher kein externer Assembler erforderlich. Das Verbinden verschiedenen Übersetzungseinheiten zu einem Programm (Linking) ist ebenfalls in den Compiler integriert. Zur Verwendung von TCC wird keinerlei externes Programm benötigt, es wird direkt von C-Code zu einem ausführbaren Programm des Zielsystems übersetzt.

- Keine getrennte Struktur: Da TCC keine Optimierung des Codes durchführt, wird auf klassische Datenstrukturen von Übersetzen verzichtet. Es gibt keinen expliziten Ableitungsbaum, keinen Syntaxbaum und keinen Zwischencode. Daher erfolgt auch keine Trennung in Front- und Backend des Compilers. Verschiedene Zielarchitekturen werden durch bedingte Kompilierung (`#ifdef`-Statements in C) umgesetzt, d.h. eine kompilierte Variante des Compilers unterstützt immer nur genau eine Zielarchitektur und ein Zielbetriebssystem.

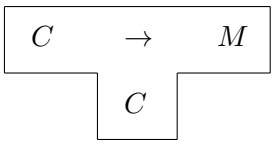


Abbildung 19: Das T-Diagramm entspricht der tatsächlichen, autarken Arbeitsweise des tcc-Compilers: Quellprogramme in der Programmiersprache C werden unmittelbar in ausführbare Maschinenprogramme übersetzt. Es erfolgt keine Trennung von Front- und Backend und es wird kein Assembler-Code erzeugt.

- One-Pass: Eine Übersetzungseinheit wird unmittelbar während des Einlesens übersetzt. Sobald genug Zeichen des Quellprogramms gelesen wurden, um eine Maschinencode-Anweisung zu erzeugen, wird diese auch im Speicher erzeugt. Bis auf zwei Ausnahmen erfolgt kein Zurückspringen im Quellprogramm. Eine Ausnahme bilden Arrays mit konstanten Initialisierung, bei denen der Programmierer die Größe nicht angibt (z.B. `int x[] = {1, 2, 3};`). Hier erfolgt ein Look-Ahead, um die Anzahl der Elemente und damit die Größe des Arrays zu bestimmen. Die zweite Ausnahme bilden Funktionsaufrufe, da hier die Übergabereihenfolge der Parameter auf dem Laufzeit-Stack unter einigen Architekturen umgekehrt zur Reihenfolge im C-Code ist.

Abbildung 19 zeigt das T-Diagramm für TCC. Die Technik des One-Pass-Übersetzens ermöglicht TCC das Verzichten auf Datenstrukturen wie Syntaxbäume und Zwischencode. Ein Nachteil ist allerdings, dass viele Optimierungen ohne diese Strukturen gar nicht möglich sind. So erzeugt TCC während des Einlesens einer Funktion bereits den Maschinencode für diese Funktion. Es kann zu einem späteren Zeitpunkt nicht festgestellt werden, ob diese Funktion überhaupt aufgerufen wird, sodass eine Dead-Code-Elimination nicht möglich ist. Weiterhin kann keine Rückwärtsanalyse durchgeführt werden, sodass keine Lebendigkeitsanalyse von Variablen und keine intelligente Registerallokation durchgeführt werden kann.

TCC verwendet im Zielcode 3 temporäre Register zur Berechnung von Ausdrücken – ist eines belegt, wird der Wert im Prozedurrahmen gesichert. Als zentrale Datenstruktur des Übersetzers wird ein Stack verwendet, der die Werte von Ausdrücken in Form von Konstanten, Registern oder Verweisen auf Einträge im Prozedurrahmen speichert. Er wird innerhalb von TCC als vStack bezeichnet.

Als weitere Datenstruktur wird ein Hash zur Verwaltung von Symbolen benutzt. In diesem werden die Bezeichner von Variablen, Funktionen usw. abgelegt und mit einer Position im vStack versehen, um in Ausdrücken auf deren Wert zugreifen zu können. In der Sprache C müssen sämtliche Bezeichner vor ihrer Verwendung deklariert werden; bei Funktionen ggf. durch Prototypen, falls die Funktion erst nach der Verwendung definiert wird. Wegen dieser Eigenschaft von C liegen sämtliche Bezeichner also bereits vor der Verwendung im vStack und ändern während ihrer Gültigkeit nicht ihre Position darin.

Ein Problem bei diesem Ansatz ist, dass die Werte von Ausdrücken unbekannt sein können, weil die Adresse eines Operanden noch unbekannt ist. Wenn in einer Funktion beispielsweise eine andere Funktion aufgerufen wird, die dem Compiler bisher nur als Prototyp bekannt

ist, so kann nicht der endgültige Maschinencode für den Funktionsaufruf emittiert werden, da die Adresse der aufgerufenen Funktion unbekannt ist. In solchen Situationen erzeugt der Compiler Maschinencode, bei dem die unbekannte Adresse als 0 codiert ist und trägt die Position der Adresse innerhalb des Maschinencodes zusammen mit dem Verweis auf das unbekannte Symbol in eine Liste ein. Nachdem das gesamte Quellprogramm eingelesen wurde und dadurch alle Adressen der Symbole bekannt sind, wird diese Liste im Rahmen des sogenannten *Symbol Patching* abgearbeitet, indem die gemerkten Stellen im Maschinencode mit den nun bekannten Adressen überschrieben werden.

Die folgenden Unterabschnitte beschreiben, wie TCC C-Code in Maschinenprogramme übersetzt. Hierbei wird darauf hingewiesen, dass der TCC-Code starke Merkmale imperativen Codes aufweist. Insbesondere werden viele globale Variablen genutzt, die Einfluss auf die Arbeitsweise von Funktionen haben. Sämtliche Funktionen sind daher grundsätzlich als zustandsbehaftet anzusehen. Mit dem Begriff „Funktion“ sind hier also Funktionen im Sinne des C-Standards und nicht Funktionen im mathematischen Sinne gemeint.

5.2.1 Lexikalische Analyse

Die lexikalische Analyse des Compilers wird durch eine Funktion namens `next` realisiert, die das nächste Token der Eingabe liest. Durch diese Funktion wird auch der Präprozessor für die Sprache C realisiert:

- falls das Token eine `#include`-Anweisung ist, so wird die eingebundene Datei gelesen und das erste Token dieser Datei ausgewertet. Erst, sobald durch weitere Aufrufe der Funktion diese Datei vollständig gelesen wurde, wird das erste Token hinter der `#include`-Anweisung zurückgegeben. Die Funktion besitzt einen internen Zustand, der die Position innerhalb der Quelldatei und in inkludierten Dateien speichert.
- falls das Token eine `#define`-Anweisung ist, mit der textuelle Substitutionen definiert werden können, so wird diese analysiert und in einer Datenstruktur gespeichert, um später prüfen zu können, ob ein substituierter Begriff erkannt wurde. Das von der Funktion gelesene Token ist dann das erste Token *hinter* der gesamten Präprozessor-Anweisung.
- falls ein substituierter Begriff gelesen wurde, so wird die Funktion rekursiv auf den substituierten Begriff angewendet, bis ein tatsächliches Token gelesen wird.

Der Wert des zuletzt gelesenen Tokens wird in der globalen Variable `tok` gespeichert. Nach einem Aufruf von `next` befindet sich hier also das nächste zu verarbeitende Token, nachdem etwaige Präprozessor-Anweisungen ausgewertet wurden. Falls es sich beim gelesenen Token um einen Bezeichner handelt, so wird in der Hash-Tabelle geprüft, ob dieser im aktuellen Scope gelesen wurde, ansonsten wird er dort eingetragen. Jeder Bezeichner enthält ein eigenes Token in einem bestimmten Wertebereich, sodass stets geprüft werden kann, ob ein Bezeichner gelesen wurde. Falls Token mit zusätzlichen Werten annotiert werden müssen, beispielsweise für Zeichenketten oder Konstanten, so befinden sich deren Werte in der globalen Variable `tokc`. Hierbei handelt es sich um eine `union`, d.h. es können sowohl Zeichenketten als auch Zahlen verschiedener Datentypen gespeichert werden. Der benutzte Datentyp hängt vom

gelesenen Token ab. Falls `tok` den Wert `TOK_STR` besitzt, so enthält `tokc.cstr` beispielsweise den Wert der Zeichenkette.

5.2.2 Syntaktische Analyse und Codeerzeugung

Das Parsen des Programms geschieht in TCC über die Technik des rekursiven Abstiegs unter Verwendung des vStacks. Den Einstieg bildet eine Funktion zum Parsen von Deklarationen, da ein C-Programm aus Sicht der Grammatik eine Folge von Deklarationen ist. Die Funktion liest in einer Schleife so lange Deklarationen ein, bis das Ende des Quellprogramms erreicht ist. Innerhalb der Funktion wird zwischen der Deklaration von globalen Variablen, Funktionsprototypen, Funktionen und Typdeklarationen (`typedef`-Anweisungen) unterschieden und rekursiv in dafür vorgesehene Funktionen abgestiegen.

Als Beispiel soll das folgende Programm übersetzt werden:

```
1  int main() {
2      int a = 4 + 2;
3      return a - 2;
4  }
```

Den Einstieg liefert die Funktion zum Parsen von Deklarationen. Als erstes wird dazu eine Funktion zum Parsen des Deklarationstyps aufgerufen. Da `int` ein Basistyp der Sprache C ist, wird dies entsprechend erkannt und im Parse-Zustand gespeichert. Da das folgende Token („main“) kein Schlüsselwort der Sprache C ist, wird es als Bezeichner erkannt und entsprechend gespeichert. Die öffnenden Klammer weist den Parser darauf hin, dass es sich um eine Funktionsdeklaration handeln muss, als nächstes werden daher die Parameter der Funktion analysiert (hier: keine Parameter). Nachdem die Funktionsdeklaration durch die schließende Klammer abgeschlossen wurde, wird der Prototyp der Funktion auf dem vStack abgelegt, um ihn für die spätere Verwendung zu speichern.

Da auf den Prototypen die öffnende, geschweifte Klammer folgt, handelt es sich nur um eine Deklaration, sondern um eine gleichzeitige Definition der Funktion. Deshalb wird nun der Prolog für die Funktion erzeugt. Die Implementierung des Prologs hängt durch bedingte Kompilierung von der gewählten Zielarchitektur ab. Im Fall von x86-Code wird der zuvor beschriebene Funktionsprolog `push ebp; mov ebp, esp; ...` zur Einrichtung eines Prozedurrahmens generiert. Zunächst erfolgt die Speicherung des Maschinencodes allerdings nur auf dem Heap – die Ausgabedatei wird erst nach der Kompilierung aller Übersetzungseinheiten erzeugt. Bemerkenswert ist, dass der erzeugte Maschinencode tatsächlich von Hand assembliert wurde und in Form von hexadezimalen Konstanten in das Zielprogramm geschrieben wird, Anweisungen wie die folgende finden sich daher überall im TCC-Code:

```
1  o(0xe58955); /* push ebp; mov ebp, esp */
```

Nachdem der Prolog erzeugt wurde, wird in eine Funktion zum Parsen von Blöcken abgestiegen, um die eigentlichen Anweisungen der Funktion zu analysieren. Zuerst werden lokale Variablendeklarationen analysiert, da diese in C vor der ersten Anweisung eines Blocks erfolgen müssen. Dazu werden analog zum Parsen der Funktionsdeklaration Name und Typ der

Variablen `a` im vStack und Hash abgelegt. Da auf die Deklaration von `a` ein Gleichheitszeichen folgt, wird Platz für die Variable im Prozedurrahmen reserviert (hier: an Position EBP - 4). Danach wird der tatsächlichen Wert für die Initialisierung analysiert. Dabei wird zunächst die 4 als konstanter Integer auf den vStack gelegt, bevor durch das Lesen des '+' eine Rotation auf dem Stack geschieht, sodass die Werte 4 und 2 oben auf dem Stack liegen und '+' wieder das aktuelle Token ist. Nun wird der Ausdruck aus dem Stack analysiert, um Operationen mit Konstanten optimieren zu können. Da beide Operanden hier konstant sind, werden die Werte 4 und 2 sowie der Operator + auf dem Stack durch das Ergebnis der konstanten Operation ersetzt, also 6. Da der Wert von `a` nun bekannt ist, wird eine Maschinencode-Anweisung erzeugt, die den Wert von `a` in den zuvor reservierten Platz im Prozedurrahmen schreibt. In TCC gibt es hierfür allerdings keine atomare Operation. Stattdessen gibt es eine Funktion, die einen Wert vom vStack in ein freies Register schreibt und eine weitere Funktion, die ein Register in den Prozedurrahmen sichern kann. In diesem Fall erzeugt die erste Funktion den Code `mov eax, 6` und die zweite Funktion den Code `mov dword ptr [ebp - 4], eax`. Ein optimierender Compiler hätte beide Operationen zusammenfassen und den Code `mov dword ptr [ebp - 4], 6` erzeugen können. Die Markierung, dass der Wert von `a` konstant ist, geht bei dieser Operation in TCC ebenfalls verloren, sodass keine Propagation der Konstante erfolgen kann. Der Grund hierfür ist, dass der Code zum Speichern des Wertes von `a` bereits erzeugt wurde. Wird später festgestellt, dass der Wert von `a` nicht benötigt wird oder ebenfalls mit konstanten Operanden verknüpft wird, so könnte der Compiler das Ablegen im Prozedurrahmen nicht rückgängig machen. Jede Anweisung im Code sorgt also unmittelbar für die Erzeugung eines entsprechenden Maschinencodes.

Die letzte Zeile der Funktion wird ähnlich analysiert: Das `return`-Schlüsselwort wird als Token erkannt und ein Ausdruck erwartet, da die Funktionsdeklaration einen Rückgabewert fordert. Dieser wird analog zur vorigen Codezeile ausgewertet, allerdings kann nun keine Optimierung von konstanten Ausdrücken durchgeführt werden, da die Information über den konstanten Wert von `a` verloren ist. Stattdessen wird Maschinencode generiert, um die Berechnung durchzuführen. Auf dem vStack liegen hier `a` und 2 sowie der Operator +. Es wird nun wieder Code erzeugt, der den Wert von `a` in ein freies Register bringt, in diesem Fall wird die Anweisung `mov eax, dword ptr [ebp - 4]` generiert. Da der zweite Operand konstant ist, wird danach die Anweisung `sub eax, 2` erzeugt.

Da die Funktion nun mit der schließenden, geschweiften Klammer endet, wird nun der Epilog der Funktion erzeugt. Insgesamt ist für den eigentlichen Code der Funktion folgender Maschinencode erzeugt worden:

```
1  mov eax, 6                      ; Laden des konstanten Wertes von a
2  mov dword ptr [ebp - 4], eax    ; Speichern im Prozedurrahmen
3  mov eax, dword ptr [ebp - 4]    ; Laden aus dem Prozedurrahmen
4  sub eax, 2                      ; Subtraktion
```

Aufgrund des One-Pass-Ansatzes kann der Compiler nicht feststellen, dass die Zeilen 2 und 3 zur Speicherung und Wiederherstellung der Variable unnötig sind. Ebenfalls kann keine Rückwärtspropagation der Konstanten durchgeführt werden, sodass die Funktion nicht insgesamt als konstant erkannt wird.

5.3 Erweiterung der C-Syntax durch eine domänenspezifische Sprache

In Abschnitt 4 wurde argumentiert, dass die Verwendung einfacher Verschleierungstechniken sinnvoll sein kann, wenn diese nicht auf das ganze Programm, sondern selektiv auf einzelne Bereiche angewendet werden. Um dies zu ermöglichen, wird die Syntax der Programmiersprache C um eine domänenspezifische Sprache erweitert, damit im Quelltext annotiert werden kann, auf welche Weise bestimmte Bereiche im Maschinencode verschleiert werden sollen. Der C-Standard hat solche Erweiterungen durch die Präprozessor-Anweisung `#pragma` bereits vorgesehen. Hinter einer solchen Anweisung darf beliebiger Text folgen, er wird implizit durch das Ende der Zeile beendet. Der C-Standard sieht vor, dass Compiler den Text hinter der `#pragma`-Anweisung ignorieren, falls sie ihn nicht als spezielle Anweisung verstehen. Compiler, die den Text als Anweisung verstehen, sollen diesen so interpretieren, dass die Semantik des Programms dadurch nicht geändert wird. Ein Quelltext mit `#pragma`-Anweisungen soll sich also nach dem Übersetzen zur Laufzeit semantisch identisch verhalten – unabhängig davon, ob der Compiler die Anweisungen verstehen konnte oder nicht. Solche Anweisungen können daher genutzt werden, um Informationen an Compiler zu geben, die sie verarbeiten können. Dabei ist gleichzeitig garantiert, dass andere Compiler keine Fehlermeldung oder Warnung erzeugen. In der Praxis werden solche Anweisungen beispielsweise genutzt, um bei Schleifen zu annotieren, dass diese vom Compiler parallelisiert umgesetzt werden dürfen.

Zur Umsetzung der diskutierten Verfahren werden daher folgende `#pragma`-Anweisungen in TCC implementiert:

- `#pragma obfuscate_string(`$s_1, \ldots, s_n$`)`
  Diese Anweisung kann genutzt werden, um Zeichenketten nach dem in Abschnitt 4.3 vorgestellten Verfahren zu verschleiern. Dazu wird eine Liste von Zeichenketten s_1 bis s_n angegeben, mit denen die nächsten n Zeichenketten verschleiert werden. Beispiel:

  ```
  1 #pragma obfuscate_string("State: Ok", "Debug Information")
  2 MessageBox(hwnd, "Invalid Serial", "Error", MB_ICONEXCLAMATION);
  ```

  Die beiden Zeichenketten, die als Parameter der `MessageBox`-Funktion dienen, werden im Maschinenprogramm durch die beiden Zeichenketten der Annotation ersetzt. Während ein Compiler, der die Annotation nicht versteht, unverschleierten Maschinencode erzeugt, soll die modifizierte TCC-Variante Code erzeugen, in dem nur die beiden Zeichenketten aus der Annotation vorkommen, wobei diese zur Laufzeit in die gewünschten, verschleierten Zeichenketten umgewandelt werden.

- `#pragma obfuscate_function(control_flow)`
  Diese Annotation kann vor der Deklaration von Funktionen angebracht werden und weist den Compiler an, den Kontrollfluss der Funktion nach dem Verfahren aus Abschnitt 4.1 mithilfe der Trampolinfunktion zu verschleiern.

- `#pragma obfuscate_function(function_calls)`
  Mit dieser Annotation, die ebenfalls vor der Deklaration angebracht werden kann, wird der Compiler angewiesen, Funktionsaufrufe innerhalb der annotierten Funktion zu verschleiern. Dazu werden die opaken Konstanten aus Abschnitt 3.5 verwendet, um zur Laufzeit die Adresse der aufgerufenen Funktion zu berechnen.

- `#pragma obfuscation_checkpoint(`c, n`)` mit $c, n \in \mathbb{N}, c \geq 0, n > 0$
  Durch diese Annotation wird der Compiler darauf hingewiesen, dass die folgende Anweisung einen Programmpunkt im Sinne des eigenen Verschleierungsverfahrens aus Abschnitt 4.2 handelt. Dabei ist n der Index des Programmpunkts innerhalb der Folge. Durch die Nutzung von c können mehrere Folgen von Programmpunkten definiert werden, um verschiedene Teile des Programms nach diesem Verfahren zu verschleiern, die zeitlich unabhängig oder unvorhersehbar voneinander ausgeführt werden. Mit c wird der Index der Programmpunktfolge angegeben, mit n der Programmpunkt innerhalb dieser Folge.

- `#pragma obfuscate_function(data_access(`c, n`))`
  Mit dieser Annotation kann eine Funktion versehen werden, um Zugriffe auf globale Daten wie Zeichenketten und globale Variablen zu verschleiern. Hierbei werden nicht die Daten selbst verschleiert, sondern der Zugriff auf die Adressen dieser Daten innerhalb der Funktion. Dazu wird das Programmpunktverfahren verwendet – mit c und n wird der zuletzt aufgerufene Programmpunkt angegeben, der vor dem Eintritt der Funktion erreicht wurde.

5.4 Implementierung der Sprache in TCC

Um die Implementierung von Verschleierungstechniken in TCC zu ermöglichen, wird der Code des Compilers um eine Schnittstelle erweitert. Diese wird in Form von Funktionen realisiert, die zu bestimmten Zeitpunkten während des Parsens und der Codererzeugung aufgerufen werden. Weiterhin wird der globale Compiler-Zustand `TCCState` um eine `struct` mit dem Namen `OBFState` zur Speicherung von Zustandsinformationen für die Verschleierung erweitert.

5.4.1 Eine flexible Schnittstelle für Verschleierungstechniken

Folgende Funktionen werden zur Umsetzung von Verschleierungstechniken hinzugefügt und im TCC-Code an den relevanten Stellen aufgerufen:

- `on_state_init`: Nachdem der Compiler seinen internen Zustand initialisiert hat, wird diese Funktion aufgerufen. Sie kann genutzt werden, um den Zustand von `OBFState` zu initialisieren.

- `on_parse_pragma`: Diese Funktion wird aufgerufen, wenn während der lexikalischen Analyse die Präprozessor-Anweisung `#pragma` im Quellprogramm gelesen wird. Dies kann genutzt werden, um Annotationen des Quellcodes zu parsen und im `OBFState` entsprechende Felder zu füllen.

- `on_pre_function`: Vor der Erzeugung des Prologs einer Funktion wird diese Funktion aufgerufen. Darin können beispielsweise Hilfsfunktionen im Maschinencode erzeugt werden, die für eine verschleierte Funktion erforderlich sind.

- `on_post_function_prolog`: Diese Funktion wird nach der Erzeugung eines Funktionsprologs aufgerufen. Sie kann genutzt werden, um Maschinencode zu erzeugen, der vor den ersten Anweisungen der Funktion ausgeführt wird, beispielsweise zur Laufzeit-Initialisierung von Verschleierungstechniken.
- `on_post_function`: Nachdem ein Funktionsepilog für eine Funktion erzeugt wurde, wird diese Funktion aufgerufen. In ihr kann beispielsweise der Zustand von `OBFState` geändert werden.
- `on_block`: Diese Funktion wird innerhalb von Anweisungsblöcke an Stellen aufgerufen, an denen Maschinencode eingefügt werden kann, ohne dass dieser Seiteneffekte auf Anweisungen des Blocks haben kann. Register und Flags können mit den entsprechenden TCC-Funktione also beliebig manipuliert werden.
- `on_string_parse`: Wenn die lexikalische Analyse eine Zeichenkette gelesen hat, wird anschließend diese Funktion aufgerufen. Die Zeichenkette kann manipuliert und ihr interner Datentyp annotiert werden, um Verschleierungsoperationen auf Zeichenketten zu ermöglichen.
- `on_post_decl_init`: Wird nach der Funktion `decl_initializer_alloc` aufgerufen, sobald das Symbol eingerichtet wurde. Hier können neu erzeugte Symbole intern annotiert werden, um Informationen zur Verschleierung für die spätere Verwendung direkt im Symbol zu speichern, beispielsweise die Information, dass eine Zeichenkette verschleiert ist und vor dem Zugriff eine Entschleierung stattfinden muss.
- `on_post_compile`: Diese Funktion wird aufgerufen, nachdem eine Übersetzungseinheit kompiliert wurde. Sie kann genutzt werden, um den Zustand von `OBFState` zurückzusetzen, soweit dies für die Übersetzung einer neuen Übersetzungseinheit erforderlich ist.
- `on_state_free`: Vor dem Aufräumen von `TCCState` wird diese Funktion aufgerufen. Sie kann genutzt werden, um `OBFState` aufzuräumen.

Der folgende Quelltext demonstriert das Zusammenspiel der Funktionen. An den mit Kommentaren markierten Stellen werden Funktionen der beschriebenen Schnittstelle aufgerufen:

```
1  /* 1 */
2  #pragma /* 2 */ obfuscate_function(control_flow)
3  void foo() { /* 3 */
4      /* 4 */
5  #   pragma /* 5 */ obfuscate_string("No Test")
6      output("Test" /* 6 */);
7      /* 7 */
8  } /* 8 */
9
10 /* 9 */
11 int main() { /* 10 */
12     /* 11 */
13     foo();
14     /* 12 */
15 } /* 13 */
16 /* 14 */
```

An Markierung 1 wird die Funktion `on_state_init` aufgerufen, falls es sich bei dieser Datei um die erste Übersetzungseinheit im Kompiliervorgang handelt. Da in der folgenden Zeile eine `#pragma`-Anweisung gelesen wurde, wird an Markierung 2 die Funktion `on_parse_pragma` aufgerufen. In dieser könnte nun die restliche Zeile geparst werden, um schließlich in `OBFState` zu speichern, dass die als nächstes gelesene Funktion einen verschleierten Kontrollfluss besitzen soll. Da in Zeile 3 eine Funktion definiert wird, folgt ein Aufruf der Funktion `on_pre_function` an Markierung 3. An dieser Stelle wurde noch keinerlei Code für die Funktion erzeugt, auch der Prolog nicht. In dieser Funktion könnte das zuvor gesetzte Flag, das die Verschleierung der nächsten Funktion vorschreibt, mit dieser Funktion verbunden werden, da erst jetzt feststeht, dass *diese* Funktion einen verschleierten Kontrollfluss besitzen soll. Es könnte beispielsweise nun ein Flag gesetzt werden, das bei der Erzeugung von Kontrollflussanweisungen wie Sprüngen dafür sorgt, dass diese verschleiert werden. An Markierung 4 werden zwei Funktionen aufgerufen: Zuerst folgt ein Aufruf von `on_post_function_prolog`, da der Prolog der Funktion zwischen Markierung 3 und 4 erzeugt wurde. Gleichzeitig kann an dieser Stelle Maschinencode eingefügt werden, der keinen Seiteneffekt auf die Anweisung der Funktion hat, weshalb auch `on_block` aufgerufen wird. An Markierung 5 wird erneut `on_parse_pragma` aufgerufen. Hier kann in `OBFState` gespeichert werden, dass die als nächstes gelesene Zeichenkette verschleiert werden soll. Die Funktion `on_string_parse` wird an Markierung 6 aufgerufen, da hier eine Zeichenkette im Quellcode gelesen wurde. Am Ende der vorigen Zeichenkette wurde sie nicht aufgerufen, da Zeichenketten innerhalb von Präprozessor-Anweisungen keine Bedeutung für das eigentliche Parsen haben. An Markierung 6 kann also erst ausgewertet werden, dass *diese* Zeichenkette wegen Markierung 5 verschleiert werden soll. Sie könnte hier also durch die verschleierte Zeichenkette ersetzt werden. Danach folgt ebenfalls an Markierung 6 ein Aufruf von `on_post_decl_init`, nachdem der Speicherplatz für die Zeichenkette im Datensegment gefüllt wurde. Durch diese Funktion könnte der für die Entschleierung nötige Schlüssel im Datensegment hinter der Zeichenkette abgelegt werden. Bei Markierung 7 handelt es sich wieder um eine sichere Stelle zum Einfügen von Maschinencode, weshalb erneut die Funktion `on_block` aufgerufen wird. Zuletzt wird an Markierung 8 die Funktion `on_post_function` aufgerufen, da nun der Maschinencode für die Funktion erzeugt wurde. Hier kann das Flag zurückgesetzt werden, das für eine Verschleierung von Kontrollflussanweisungen sorgt.

Die restlichen Markierungen entsprechen den vorigen: Markierung 9 entspricht Markierung 3, Markierung 10 entspricht Markierung 3 und so weiter. Bei Markierung 14 wird neben `on_post_function` auch `on_post_compile` aufgerufen, da die Übersetzungseinheit hier endet. Falls es sich um die letzte Übersetzungseinheit im Kompiliervorgang handelte, wird zusätzlich `on_state_free` aufgerufen.

5.4.2 Implementierung der Zeichenkettenverschleierung

Einzelne Zeichenketten können mithilfe des Verfahrens aus Abschnitt 4.3 auf Seite 60 verschleiert werden. Dazu wird eine Annotation *vor* der zu verschleiernden Zeichenkette angebracht:

```
1 #pragma obfuscate_string("State: Ok", "Debug Information")
2 MessageBox(hwnd, "Invalid Serial", "Error", MB_ICONEXCLAMATION);
```

In diesem Beispiel soll statt der Zeichenkette „Invalid Serial“ die unverdächtige Zeichenkette „State: Ok“ auftauchen. Ebenfalls soll statt „Error“ an dieser Stelle die Zeichenkette „Debug Information“ im Maschinencode abgelegt werden. Die tatsächlich zur Laufzeit auftauchenden Zeichenketten werden als *verdächtige Zeichenketten* bezeichnet, die im Maschinencode auftauchenden Zeichenketten als *unverdächtige Zeichenketten*.

Das Verfahren wird unter Verwendung der oben beschriebenen Schnittstelle folgendermaßen implementiert:

- im globalen Zustand `OBFState` wird eine FIFO durch eine verkettete Liste implementiert, um die unverdächtigen Strings speichern zu können, die später im Maschinencode sichtbar sein sollen. Die FIFO ermöglicht die Operationen `front` zum Zugriff auf das erste Element, `push_back` zum Anfügen eines neuen Elements am Ende sowie `del_front` zum Löschen des ersten Elements. Die Operation `is_empty` zeigt an, ob die FIFO leer ist.

- wenn in `on_parse_pragma` die Anweisung `#pragma obfuscate_string(`$s_1, \ldots, s_n$`)` erkannt wird, so werden die Zeichenketten s_1 bis s_n an das Ende der FIFO eingefügt. Es erfolgt zunächst also nur eine Speicherung – auf die zu verschleiernden Zeichenketten kann hier noch nicht zugegriffen werden, da dies gegen die One-Pass-Architektur des Compilers verstoßen würde.

- nach jedem Parsen einer Zeichenkette wird in `on_string_parse` geprüft, ob die FIFO ein Element enthält. Wenn dies der Fall ist, bedeutet das, dass an einer Stelle vor dem Auftreten der Zeichenkette eine Anweisung zur Verschleierung der Zeichenkette auftrat. Demzufolge soll die gerade geparste Zeichenkette verschleiert werden. Dazu wird das oben beschrieben Verfahren eingesetzt: die zu verschleiernde Zeichenkette sowie die unverdächtige Zeichenkette werden auf die nächstgrößere gemeinsame Länge gebracht, die ein Vielfaches von 4 ist. Über xor-Operationen wird dann der Schlüssel ausgerechnet, der die unverdächtige Zeichenkette in die verschleierte Zeichenkette überführt. Zuletzt wird die gerade geparste Zeichenkette durch die Konkatenation von unverdächtiger Zeichenkette und Schlüssel ersetzt. Dadurch taucht die verschleierte Zeichenkette nicht im Programm auf. Im Typ der Zeichenkette wird annotiert, dass die Zeichenkette verschleiert ist. Die unverdächtige Zeichenkette wird der FIFO entnommen, da sie nun nicht mehr benötigt wird.

- wird in `on_obf_string_access` festgestellt, dass auf eine verschleierte Zeichenkette zugegriffen werden soll, so wird im Prozedurrahmen Platz reserviert, um die Zeichenkette zu entschlüsseln. Die Entschlüsselung wird durch eine vordefinierte Funktion durchgeführt, die beim Übersetzen in das Programm gelinkt wird. In dieser wird die Zeichenkette in Blöcken von je 4 Bytes entschlüsselt, sodass sie danach unverschleiert im Prozedurrahmen liegt. Die Adresse im Prozedurrahmen wird nun beim Zugriff auf die Zeichenkette anstelle der Adresse der unverdächtigen Zeichenkette genutzt.

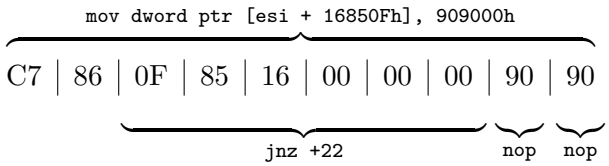


Abbildung 20: Der Opcode der Anweisung `mov dword ptr [esi + x], y` besteht aus 10 Bytes. Die hinteren 8 Bytes codieren dabei die Werte x und y, sodass sie frei gewählt werden können. In diesem Beispiel wurden x und y so gewählt, dass darin die 6 Bytes des Opcodes der Anweisung `jnz +22` codiert wurden. Die verbleibenden 2 Bytes wurden jeweils auf den Wert 90 gesetzt, der dem Opcode der effektlosen `nop`-Anweisung entspricht. Aus Sicht des Disassemblers werden die 10 Bytes als zusammenhängende Anweisung decodiert (Klammer oben). Zur Laufzeit werden allerdings durch die Trampolinfunktion die ersten beiden Bytes des Opcodes übersprungen, sodass der Prozessor die drei für den Disassembler versteckten Anweisungen (Klammern unten) ausführt.

5.4.3 Implementierung der Kontrollflussverschleierung

Der Kontrollfluss kann über das modifizierte Verfahren von Balachandran mithilfe der Trampolinfunktion (siehe Abschnitt 4.1 auf Seite 48) verschleiert werden. Dazu kann eine Annotation vor der Definition angebracht werden, um zu markieren, dass die folgende Funktion einen verschleierten Kontrollfluss im Maschinencode besitzen soll:

```
1 #pragma obfuscate_function(control_flow)
2 void test(int n) {
3     int i;
4     for(i = 0; i < n; i++) {
5         ...
6     }
7 }
```

In diesem Beispiel soll der Kontrollfluss der Funktion `test` so verschleiert werden, dass die Schleife im Maschinencode nicht als solche erkannt wird, es also keine Zyklen im Kontrollflussgraph gibt.

Die Umsetzung wurde folgendermaßen vorgenommen:

- wenn in `on_parse_pragma` die Anweisung `#pragma obfuscate_function`$(p_1, \ldots, p_n)$ erkannt wird und einer der Ausdrücke p_1 bis p_n dabei `control_flow` entspricht, so wird im globalen Zustand in einem Flag gespeichert, dass die nächste Funktion einen verschleierten Kontrollfluss besitzen soll.
- in `on_pre_function` wird das Flag ausgewertet – ist es gesetzt, so ist an dieser Stelle klar, dass die Annotation sich auf die gerade geparste Funktionsdefinition beziehen muss. Ist dies der Fall, so wird eine Trampolinfunktion erzeugt, die von dieser Funktion zur Verschleierung genutzt werden kann. Da diese Funktion nur wenige Anweisungen besitzt (vgl. Abschnitt 4.1), wird für jede Funktion mit verschleiertem Kontrollfluss eine eigene Trampolinfunktion generiert. Auf diese Weise wird die Heimlichkeit des

Verfahrens erhöht, da es dann keine zentrale Trampolinfunktion gibt, deren Kenntnis sofort zur Aufdeckung aller verschleierten Sprünge genutzt werden könnte. Die Adresse der generierten Trampolinfunktion wird im globalen Zustand gespeichert.

- wenn innerhalb einer Funktion ein Sprung erzeugt werden soll und das Flag gesetzt ist, so wird der aus den 6 Bytes b_1 bis b_6 bestehende Opcode (2-Byte Sprung-Opcode und 4-Byte relative Zieladresse) in einer Anweisung der Form `mov [esi + x], y` versteckt. Der Opcode dieser Anweisung ist 10 Bytes lang: `C7 86 X4 X3 X2 X1 Y4 Y3 Y2 Y1`. Die letzten 8 Bytes können dabei frei gewählt werden, sie codieren die Werte für x und y. In ihnen kann also die gesamte zu verschleiernde Sprunganweisung versteckt werden, indem die Werte b_1 bis b_6 an die Positionen für die codierten Werte von x und y geschrieben werden. Die verbleibenden 2 Bytes werden mit dem Opcode `90` gefüllt, der einer effektlosen `nop`-Anweisung entspricht.

  Soll beispielsweise die Sprunganweisung `jnz +22` verschleiert werden, die im Fall eines gesetzen Zero-Flags um 22 Bytes im Maschinencode springt, so wird der Opcode der Anweisung (`0F 85 16 00 00 00`) in der `mov`-Anweisung versteckt: `C7 86 0F 85 16 00 00 00 90 90` entspricht der Anweisung `mov dword ptr [esi + 16850Fh], 909000h`. Diese Anweisung ist unverdächtig und an jeder Stelle im Programmcode plausibel – um zu prüfen, dass sie auf eine ungültige Adresse verweist, müsste der Inhalt des Registers `ESI` bekannt sein. Damit zur Laufzeit nicht die `mov`-Anweisung, sondern der darin verschleierte Sprung ausgeführt wird, folgt ein Aufruf der Trampolinfunktion vor dem Opcode der `mov`-Anweisung. Der Aufruf der Trampolinfunktion sorgt dafür, dass der Rücksprung um 2 Bytes verschoben ist – die ersten 2 Bytes der `mov`-Anweisung werden zur Laufzeit also übersprungen, sodass stattdessen der darin getarnte Sprung gefolgt von zwei `nop`-Anweisungen ausgeführt wird (vgl. Abbildung 20).

- wird eine Funktion im Quelltext beendet, so wird in `on_post_function` das Flag gelöscht, das die Verschleierung der Funktion markiert hat. Auf diese Weise wird nur die Funktion verschleiert, die unmittelbar auf die Annotation folgt.

5.4.4 Implementierung der opaken Konstanten

Das Verfahren zur Erzeugung von verschleierten Konstanten aus Abschnitt 3.5 auf Seite 43 ist sehr aufwändig, was die benötigte Laufzeit betrifft, da für jedes Bit einer verschleierten Konstante eine SAT-Formel zur Laufzeit ausgewertet werden muss. Es wird daher nur verwendet, um die Adressen von Funktionsaufrufen zu verschleiern, da ein Funktionsaufruf ohnehin eine vergleichsweise langsame Operation ist. Da die durch TCC erzeugten Funktionsaufrufe die Form `call <adresse>` haben, wird die Adresse als opake Konstante in einem freien Register r berechnet und anschließend der Code `call r` erzeugt, um die Funktion indirekt aufzurufen.

Die Annotation wird folgendermaßen vorgenommen:

```
1 #pragma obfuscate_function(function_calls)
2 void test() {
3   f();
```

```
4 }
```

Die Annotation gibt an, dass Funktionsaufrufe innerhalb der Funktion `test` verschleiert werden sollen. Die Adresse von f wird also als opake Konstante zur Laufzeit berechnet, in einem Register abgelegt und eine indirekte `call`-Anweisung auf das Register erzeugt, um den Aufruf von f zu verschleiern. Der Angreifer erfährt an dieser Stelle zwar, dass eine Funktion aufgerufen wird, allerdings kann er nicht unmittelbar nachvollziehen, um welche Funktion es sich handelt.

Zur Umsetzung wurde wieder die oben beschriebene Schnittstelle verwendet:

- wenn in `on_parse_pragma` die Anweisung `#pragma obfuscate_function`$(p_1, \ldots, p_n)$ erkannt wird und einer der Ausdrücke p_1 bis p_n dabei `function_calls` entspricht, so wird im globalen Zustand in einem Flag gespeichert, dass Funktionsaufrufe innerhalb der folgenden Funktion verschleiert werden sollen.
- ist das Flag gesetzt und wird ein Funktionsaufruf der Funktion mit der Adresse a geparst, so wird statt der Anweisung `call a` Code erzeugt, der die Adresse `a` zur Laufzeit als opake Konstante berechnet und in einem freien Register `r` ablegt. Danach wird via `call r` die Funktion indirekt aufgerufen.
- wird eine Funktion im Quelltext beendet, so wird in `on_post_function` das Flag gelöscht, das die Verschleierung der Funktion markiert hat. Auf diese Weise wird nur die Funktion verschleiert, die unmittelbar auf die Annotation folgt.

Um eine verschleierte Konstante c zu erzeugen, wird zunächst eine 32-Bit-Zufallskonstante erzeugt, die dem Compiler bekannt ist und die vom Zielprogramm zur Laufzeit berechnet wird. Es wäre auch möglich, direkt den Wert von c zur Laufzeit zu berechnen – später wird begründet, warum zunächst eine Zufallszahl erzeugt wird, aus der dann c berechnet wird.

Den Pseudocode für den Algorithmus zur Erzeugung des Zufallswertes zeigt Algorithmus 21: Mit einem Zufallsgenerator wird ein zufälliges Bit erzeugt. Ist dieses 0, so wird eine unerfüllbare SAT-Instanz nach dem in Abschnitt 3.5 beschriebenen Verfahren erzeugt. Ist das Zufallsbit hingegen 1, so wird eine zufällige Variablenbelegung erzeugt und für diese eine erfüllende SAT-Instanz erzeugt. Die Klauseln und die Variablen werden im Zielprogramm gespeichert. Im Falle eines 0-Bits wird eine zufällige Adresse im unitialisierten BSS-Segment des Programms gewählt, damit die Belegung der Variablen nicht bekannt ist, ohne das Programm zu starten oder zu simulieren. Die tatsächliche Belegung zur Laufzeit ist für das Verfahren irrelevant, da die SAT-Instanz in diesem Fall von keiner Belegung erfüllt wird. Die Variable `mask` enthält nach dem Lauf des Algorithmus die erzeugte, zufällige Konstante.

Soll nun eine Konstante c zur Laufzeit berechnet werden, so erzeugt der Compiler den Code von Algorithmus 22 im Maschinencode, der mit den generierten Klausen und Variablen zur Laufzeit den Wert von `mask` berechnet, indem eine Funktion zur Auswertung von SAT-Instanzen zur Laufzeit für jedes Bit aufgerufen wird. Die erfüllbaren SAT-Instanzen erzeugen mit den gegebenen Variablen die 1-Bits von `mask`, die unerfüllbaren Instanzen erzeugen mit den Laufzeit-Zufallsdaten die 0-Bits. Nachdem der Wert von `mask` durch die 32 SAT-Auswertungen in einem Register liegt, kann der Compiler Maschinencode erzeugen, der das Register durch

Algorithmus 21 createOpaqueConstant

```
mask ← 0
clauses ← emptyList()
varPointers ← emptyList()
for i = 1 to 32 do
    if randomBool() = false then                          // Erzeuge 0-bit
        curVars ← randomBSSPointer()                      // Zufaellige Laufzeitdaten
        curClauses ← createUnsatInstance()
        mask ← leftShift(mask, 1)
    else                                                  // Erzeuge 1-bit
        curVars ← randomArray()                           // Variablen mit zufaelligen Werten
        curClauses ← createSatInstance(curVars)
        mask ← bitOr(leftShift(mask, 1), 1)
    end if
    append(clauses, curClauses)
    append(varPointers, curVars)
end for
```

Algorithmus 22 evalSAT(c, clauses, vars)

```
res ← 0
for i = 1 to 32 do
    if evalClause(clauses[i], vars[i]) then               // 1-Bit
        res ← bitOr(leftShift(res, 1), 1)
    else                                                  // 0-Bit
        res ← leftShift(res, 1)
    end if
end for
return res ⊕ c
```

xor-Verknüpfung mit dem Wert `mask` $\oplus$ c von `mask` nach c überführt, denn es gilt `mask` $\oplus$ (`mask` $\oplus$ c) = c.

Durch die One-Pass-Architektur des Compilers gestaltet sich die Verschleierung von Funktionsaufrufen schwierig. Bei Funktionsaufrufen ist die tatsächliche Adresse der Funktion im virtuellen Adressraum noch nicht bekannt, da die Funktion erst später im Quelltext auftauchen kann und noch kein Code für diese erzeugt wurde. Aus diesem Grund erzeugt TCC statt der Anweisung `call a` zunächst die Anweisung `call 0`, d.h. eine `call`-Anweisung mit dem Parameter Null. In einer Datenstruktur wird vermerkt, dass diese 4 Null-Bytes später durch die tatsächliche Adresse der aufgerufenen Funktion ersetzt werden soll. Dies geschieht als letzte Phase des Übersetzers beim Schreiben des Maschinencodes in eine ausführbare Datei und wird als *Symbol Patching* bezeichnet. Da die Adresse der Funktion für das Verfahren aber zur Laufzeit berechnet werden und trotzdem nicht im Maschinencode auftauchen soll, musste das Symbol Patching des Übersetzers verändert werden: Anstatt die Null-Bytes des Funktionsaufrufs während des Patchings mit der korrekten Adresse zu überschreiben, wird

nun eine xor-Verknüpfung zwischen den Null-Bytes und der korrekten Adresse durchgeführt. Auf diese Weise wird für Null-Bytes weiterhin die korrekte Adresse geschrieben, da Null das neutrale Element der xor-Gruppe ist (vgl. Abschnitt 4.2.2 auf Seite 52). Allerdings können nun anstelle von Null-Bytes auch 4 andere Bytes geschrieben werden, die beim Symbol Patching dann mit der endgültigen Adresse xor-verknüpft werden. Diese Änderung wird genutzt, um in Algorithmus 22 die xor-Verknüpfung am Ende so zu ändern, dass statt der Konstante c der Wert von mask eingesetzt wird. Dadurch würde zur Laufzeit der Wert 0 berechnet werden, da der Wert von `mask` berechnet wird und dann mit demselben Wert verknüpft wird. Allerdings wird die Konstante der xor-Verknüpfung in die Symbol-Patching-Tabelle eingetragen, sodass der Compiler in der letzten Phase den Wert von `mask` an dieser Stelle mit der Adresse a der aufzurufenden Funktion verknüpft. Die letzte Anweisung wird also beim Symbol Patching zu `res` $\oplus$ `mask` $\oplus$ `a` modifiziert. Zur Laufzeit entsteht daraus der Wer `mask` $\oplus$ (`mask` $\oplus$ a) $= a$, also die korrekte Adresse der Funktion. Aus diesem Grund wurde das Verfahren wie oben erwähnt so implementiert, dass zunächst ein Zufallswert erzeugt wird.

5.4.5 Implementierung des Programmpunktverfahrens

Für die Umsetzung des Programmpunktverfahrens aus Abschnitt 4.2 auf Seite 50 sind zwei neue Annotationsarten erforderlich:

```
1  int main() {
2  #  pragma obfuscation_checkpoint(0, 1)
3     protected_a();
4  #  pragma obfuscation_checkpoint(0, 2)
5     return 0;
6  }
7
8  #pragma obfuscate_function(data_access(1, 1))
9  void protected_b(char *data) {
10   printf("Data: %s\n", data);
11 }
12
13 #pragma obfuscate_function(data_access(0, 1))
14 void protected_a() {
15 #  pragma obfuscation_checkpoint(1, 1)
16    protected_a("Test");
17 #  pragma obfuscation_checkpoint(1, 2)
18 }
```

Die Annotationen in den Zeilen 2 und 4 sowie 15 und 17 teilen dem Compiler mit, in welcher Reihenfolge die durch diese Annotationen definierten Programmpunkte zur Laufzeit durchlaufen werden. Es gibt in diesem Beispiel zwei unabhängige Programmpunktläufe: Zeile 2 und 4 definieren Programmpunkte für Lauf 0 und Zeile 10 und 12 für Lauf 1. Die Unterstützung für unabhängige Läufe wurde implementiert, damit verschiedene Bereiche des Programms mit diesem Verfahren geschützt werden können, falls die Reihenfolge der Bereichen untereinander nicht deterministisch ist. In diesem Fall kann der Programmierer für jeden Teilbereich des Programms einen eigenen Lauf mit Programmpunkten vorsehen, in dem die Reihenfolge der Programmpunkte immer identisch ist. Der Programmierer sichert dem Compiler im obigen

Algorithmus 23 getRandom(c, p)

```
cycle ← lookupCycle(c)                          // Daten für Laufindex nachschlagen
r_{c_p} ← lookupRandom(cycle, p)                // Zufallszahl nachschlagen
if r_{c_p} = ⊥ then                             // Zufallszahl noch nicht bestimmt
    r_{c_p} ← createRandom()                    // Zufallszahl erzeugen
    storeRandom(cycle, p, r_{c_p})              // Zufallszahl speichern
end if
return r_{c_p}
```

Algorithmus 24 getKey(c, p)

```
res ← 0
for i = 0 to p do
    r_{c_i} ← getRandom(c, i)                   // Wird erzeugt oder nachgeschlagen
    res ← res ⊕ r_{c_i}
end for
return res
```

Beispiel also zu, dass Zeile 2 vor Zeile 4 sowie Zeile 15 vor Zeile 17 ausgeführt wird. Über das zeitliche Verhältnis der beiden Läufe wird keine Aussage getroffen, sie sind unabhängig voneinander und dürfen sich zeitlich überlappen. Weiteres Wissen über die Programmpunkte wird durch die Annotationen in den Zeilen 8 und 13 zugesichert – hierdurch sichert der Programmierer dem Compiler zu, dass der jeweils angegebene Programmpunkt *vor* dem Aufruf der Funktion als letztes erreicht wurde. Diese Information kann der Compiler nutzen, um Zugriffe auf Adressen des Programms nach dem beschrieben Verfahren zu verschleiern, beispielsweise den Zugriff auf die Adresse der Zeichenketten in Zeilen 10 und 16. In den Tupeln (c, p) der Programmpunkte wird c als *Laufindex* und p als *Programmpunktindex* innerhalb des Laufes bezeichnet.

Eine Besonderheit im obigen Beispiel ist, dass der Programmpunkt $(1, 1)$ in Zeile 8 bereits zum Schutz der Funktion verwendet werden soll, obwohl dieser erst in Zeile 15 definiert wird. Dies stellt wegen der One-Pass-Architektur von TCC eine Herausforderung bei der Implementierung dar. Gemäß dem Verfahren wird die mit dem Programmpunkt $(1, 1)$ assoziierte Zufallszahl zum Verschleierung des Zugriffs auf die Zeichenkette in Zeile 10 benötigt, obwohl sie eigentlich erst in Zeile 15 berechnet würde. Zur Lösung wird eine Datenstruktur zur Verwaltung der Zufallszahlen eingeführt. In dieser Datenstruktur wird für jeden Laufindex gespeichert, welche Zufallszahlen für die zugehörigen Programmpunktindizes des Laufs verwendet wurden. Weiterhin wird dort die Adresse der globalen Variable s_c für jeden Laufindex gespeichert, da jeder Lauf für das Verfahren wegen der Unabhängigkeit der Läufe eine eigene Variable benötigt. Die Datenstruktur bietet die Möglichkeit, über eine Operation die Zufallszahl für ein Programmpunkttupel zu erhalten. Bei der ersten Anfrage an ein Tupel wird die Zufallszahl erzeugt und gespeichert, sodass weitere Anfragen an dasselbe Tupel die identische Zufallszahl liefern (siehe Algorithmus 23). Dies wird innerhalb der Datenstruktur in einer verketteten Liste gespeichert. Eine weitere Operation berechnet die xor-Verknüpfung der Zufallszahlen aller Programmpunkte eines Laufs bis zum einem Maximalindex (siehe Algorithmus 24).

Mit dieser Datenstruktur kann das Verfahren mithilfe der oben beschriebenen Schnittstelle umgesetzt werden:

- im globalen Zustand `OBFState` wird die Datenstruktur für die Verwaltung der Programmpunkte angelegt.
- wenn in `on_parse_pragma` die Anweisung `#pragma obfuscate_function(`$p_1, \ldots, p_n$`)` erkannt wird und einer der Ausdrücke p_1 bis p_n dabei `data_access(c, p)` entspricht, so wird im globalen Zustand gespeichert, dass Zugriffe auf Adressen im Datensegment über das Programmpunktverfahren geschützt werden sollen und dass der letzte Programmpunkt dem Tupel (c, p) entspricht. Weiterhin wird ein Flag gesetzt, um zu markieren, dass die folgende Funktion auf diese Weise verschleiert werden soll.
- ist das Flag zur Verschleierung der Funktion gesetzt, so wird mithilfe der Datenstruktur in `on_pre_function` geprüft, ob für den Laufindex c bereits ein Speicherort für die im Verfahren beschriebene globale Variable s_c existiert. Falls dies nicht der Fall ist, so wird dieser im Datensegment eingerichtet und die Adresse in der Datenstruktur gespeichert. Dieser Schritt ist nötig, falls eine Funktion mit dem Verfahren geschützt werden soll, bevor der Compiler die zugehörigen Programmpunkte gelesen hat. Dies ist im Beispiel in Zeile 8 der Fall, der Compiler wird also bereits an dieser Stelle die globale Variable reservieren.
- wenn in `on_parse_pragma` die Anweisung `#pragma obfuscation_checkpoint(c, p)` erkannt wird, so werden c und p im globalen Zustand gespeichert und ein Flag wird gesetzt, das eine Einfügung des Programmpunkts vor der nächsten Anweisung fordert. Da der Präprozessor von TCC in die lexikalische Analyse integriert ist, kann leider nicht direkt der Code zum Aktualisieren des Programmcodes erzeugt werden. Die Erzeugung des tatsächlichen Codes muss also bis zur nächsten geparsten Anweisung delegiert werden. Falls zwei Programmpunkte ohne Anweisung dazwischen direkt hintereinander liegen, so wird eine Fehlermeldung ausgegeben.
- wird eine Funktion im Quelltext beendet, so wird in `on_post_function` das Flag gelöscht, das die Verschleierung der Funktion markiert hat. Auf diese Weise wird nur die Funktion verschleiert, die unmittelbar auf die Annotation folgt.

Die Erzeugung des Codes zur Aktualisierung geschieht in der Schnittstellenfunktion `on_block`: Wird diese aufgerufen, während im globalen Zustand markiert ist, dass ein Programmpunkt an der nächsten Anweisung erzeugt werden soll, so wird das Flag gelöscht und der Code für den Programmpunkt wird erzeugt. Dazu wird mithilfe der Datenstruktur der Wert der Zufallszahl für den Programmpunktindex ermittelt. Nun kann der Maschinencode erzeugt werden, der die zugehörige globale Variable s_c durch eine xor-Verknüfung mit der Zufallszahl aktualisiert. Der erzeugte Maschinencode-Befehl hat also folgende Struktur, um die globale Variable s_c zu aktualisieren: `xor dword ptr[lookupGlobalS(c)], getRandom(c, p)`

Für die Umsetzung der eigentlichen Verschleierung wurde der Compiler so modifiziert, dass vor der Erzeugung von Maschinencode zum Laden von globalen Daten aus dem Datensegment zuerst geprüft wird, ob das Flag zur Verschleierung dieser Zugriffe gesetzt ist. Falls dies der Fall ist, so erzeugt der Compiler stattdessen Code, der die verschleierte Adresse der

Daten in ein Register lädt. Die verschleierte Adresse kann der Compiler an dieser Stelle mithilfe der Datenstruktur berechnen, indem die xor-Verknüpfung aller Zufallszahlen des Laufindexes c bis zum Programmpunktindex p berechnet werden. Dieser Wert wird analog zum vorigen Abschnitt im Symbol Patching mit der tatsächlichen Adresse xor-verknüpft, sodass die Adresse dadurch verschleiert im Maschinencode steht. Hinter der Anweisung zum Laden der verschleierten Adresse generiert der Compiler einen Maschinencode-Befehl, der die verschleierte Adresse mit dem Wert der zugehörigen Laufzeitvariable s_c xor-verknüpft. Das Verfahren sowie die Annotationen sichern zu, dass der Wert der Variable s_c sowie die xor-Verknüpfung der Zufallszahlen aller vorigen Programmpunkte identisch sind. Deshalb wird das Register, in das die verschleierte Konstante geschrieben wurde, mit dem Laufzeitwert der globalen Variable s_c entschleiert, sodass zur Laufzeit die unverschleierte Adresse im Register berechnet wird.

6 Evaluation

Um die Praxistauglichkeit des modifizierten Compilers und der Annotationssprache zu untersuchen, soll zunächst die Beispielanwendung geschützt werden. Anschließend wird allgemeiner untersucht, welche Auswirkungen die einzelnen Verfahren auf die Laufzeitgeschwindigkeit und Zielprogrammgröße haben.

6.1 Schutz der Beispielanwendung

Die Beispielanwendung soll mithilfe des modifizierten Compilers gegen Reverse Engineering geschützt werden. Dazu soll der Hochsprachencode des Programms nicht geändert werden, der Schutz soll allein durch die vorgestellten Annotationen automatisiert vom Compiler umgesetzt werden. Der Quellcode der Beispielanwendung befindet sich in Anhang B.1 auf Seite 96. Es folgt eine kurze Beschreibung der Funktionsweise:

- die Funktion `checkSerial` in Zeile 2 prüft eine Kombination aus Name und Seriennummer auf ihre Gültigkeit. Dazu wird die Seriennummer als hexadezimale Zahl interpretiert und mit dem Ergebnis einer Berechnung verglichen, die sich aus den ASCII-Werten des eingegebenes Namens ergibt.
- wann immer ein Ereignis im Dialog zur Registrierung der Anwendung auftritt – beispielsweise das Klicken des Benutzers auf einen Button – so wird vom Betriebssystem die Funktion `RegisterProc` in Zeile 22 aufgerufen. Der Parameter `msg` informiert über die Art des Ereignisses, die beiden anderen Parameter liefern Details zum Ereignis, die von der Art der Nachricht abhängen. Der Code ab Zeile `27` wird ausgeführt, wenn der Benutzer auf den Button „OK“ klickt, um die Anwendung mittels Name und Seriennummer freizuschalten. Hier wird die Funktion `checkSerial` zur Prüfung verwendet, anschließend wird der Erfolg bzw. Misserfolg angezeigt und der Dialog geschlossen. Hierbei wird auch der Erfolg der Registrierung zurückgegeben.
- die Funktion `onMenuRun` in Zeile 46 wird aufgerufen, wenn der Benutzer das eingegebene Lua-Skript ausführen möchte. In Zeile 51 wird geprüft, ob die Ausführung verweigert werden muss, weil das Programm nicht freigeschaltet und die Länge zu lang ist. Dies wird mit einer Fehlermeldung angezeigt.
- wenn ein Ereignis im Hauptfenster der Anwendung auftritt, so wird dies der Funktion `WndProc` in Zeile 70 übergeben. Die Parameter sind analog zu denen in der Funktion für den Dialog zur Registrierung der Anwendung. Eine Besonderheit ist, dass die Nachricht `WM_CREATE` bereits *während* des Erstellens des Fensters durch das Betriebssystems aufgerufen wird (Zeile 78).
- Zeile 122 bildet mit der Funktion `WinMain` den Einstiegspunkt in das Programm. Hier wird das Fenster erzeugt (Zeile 150) und angezeigt (Zeile 162). Die Schleife in Zeile 170 prüft schließlich, ob das Betriebssystem Nachrichten für das Fenster hat und leitet diese gegebenenfalls an die Funktion `WndProc` weiter. Die Schleife wird erst verlassen,

wenn das Fenster geschlossen wird, aus Sicht des Programms handelt es sich also um eine Endlosschleife.

Im Programm sollten also zwei Schutzgüter unabhängig voneinander geschützt werden: Einerseits sollte es dem Angreifer nicht möglich sein, das Programm ohne gültige Seriennummer freizuschalten oder sich selbst gültige Seriennummern zu erzeugen. Andererseits sollte es nicht möglich sein, das Programm ohne Freischaltung im vollen Umfang verwenden zu können. In der ungeschützten Variante ist beides möglich – es wurde bereits gezeigt, wie der Angreifer sich gültige Seriennummern erzeugen kann. Die Nutzung ohne Freischaltung wäre ebenfalls möglich, indem die Prüfung in Zeile 51 durch den Angreifer entfernt wird. Zur Umsetzung des Schutzes wird das Programm folgendermaßen geändert (vgl. Quellcode der geschützten Variante in Abschnitt B.2 auf Seite 100)

- damit der Angreifer den Maschinencode zur Prüfung der Seriennummer nicht mehr nachvollziehen kann, soll der Kontrollfluss dieser Funktion verschleiert werden. Gleichzeitig soll das Auffinden dieser Funktion erschwert werden, indem verdächtige Funktionsaufrufe wie `strlen` und `strtoul` nicht mit diesem Teil des Maschinencodes in Verbindung gebracht werden können. Deshalb wird die Funktion so annotiert, dass Kontrollfluss und Funktionsaufrufe verschleiert werden (Zeile 2).
- in der Funktion `RegisterProc` (Zeile 23) befinden sich viele verdächtige Zeichenketten. Der Zugriff auf diese soll geschützt werden, damit der Angreifer diese nicht mit der Funktion in Verbindung bringen kann. Dazu soll das Programmpunktverfahren genutzt werden. Folgende Programmpunkte werden im Programm identifiziert:
  - beim Start des Programms wird das Fenster registriert und vorbereitet. Hierbei kann es Fehler geben, die zum vorzeitigen Beenden des Programms führen. Da die geschützte Funktion aber nur im Erfolgsfall aufgerufen werden kann, ist garantiert, dass zur Laufzeit die eingefügten Programmpunkte 1 und 2 in den Zeilen 134 und 149 garantiert in dieser Reihenfolge aufgerufen wurden, falls die geschützte Funktion betreten wird.
  - In Zeile 152 folgt ein Aufruf von `CreateWindowEx` zur Erzeugung des Fensters. Wie oben beschrieben, sorgt dieser Funktionsaufruf dafür, dass das Betriebssystem die Funktion `WndProc` in Zeile 70 mit dem Parameter `WM_CREATE` aufruft, weshalb hier Programmpunkt 3 eingefügt wird (Zeile 80). Dies ist für den Angreifer selbst bei manueller Analyse schwer nachzuvollziehen, da er hierzu den Maschinencode als zugehörige Fensterfunktion des erzeugten Fensters identifizieren müsste. Ein Disassembler kann die Reihenfolge dieser Aufrufe auf keinen Fall ermitteln, ohne die Semantik der Funktionen des Betriebssystems zu kennen.
  - nachdem das Fenster angezeigt wurde, folgt Programmpunkt 4 in Zeile 167. Dies ist der letzte, der vor der Endlosschleife des Programms ausgeführt wird, weshalb er auch zum Schutz der Funktion genutzt wird. Programmpunkt 5 wurde nach der Endlosschleife eingeführt und wird zur Laufzeit erst unmittelbar vor dem Beenden des Programms erreicht. Auch diese Semantik der Schleife kann ein Disassembler nicht nachvollziehen.

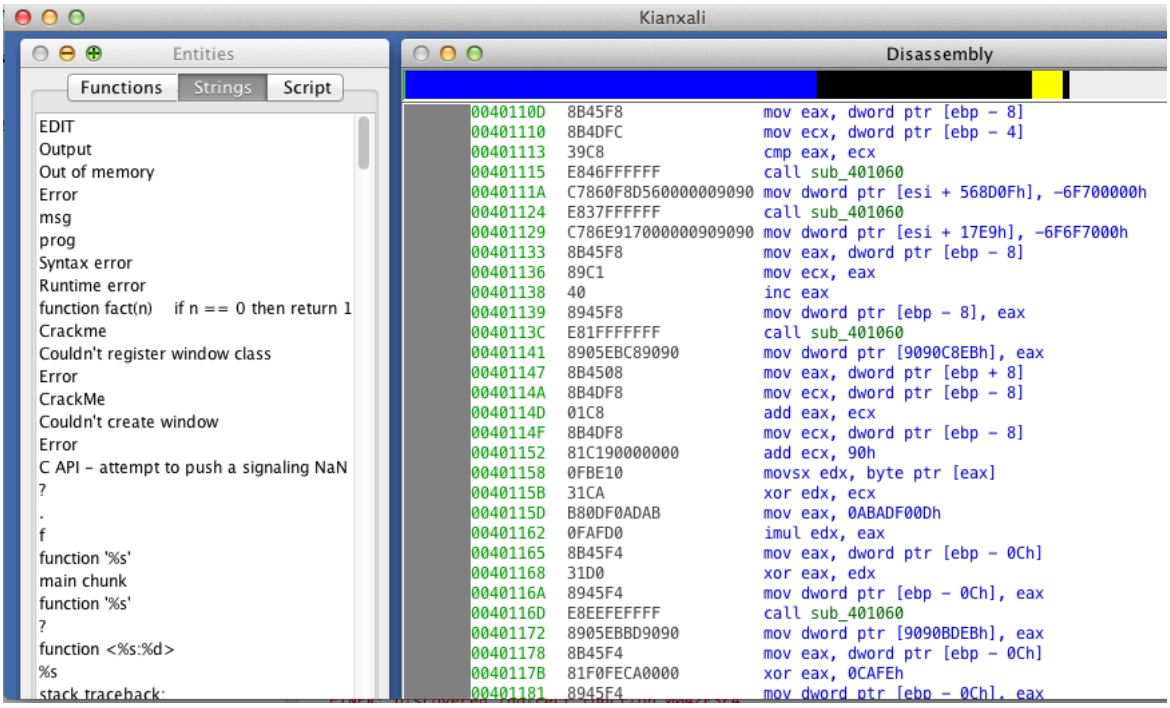


Abbildung 25: Die Abbildung zeigt das verschleierte Programm in Kianxali. Links ist erkennbar, dass keine verdächtigen Zeichenketten mehr angezeigt werden. Gleichzeitig werden alle anderen Zeichenketten aber noch angezeigt, sodass dem Angreifer vorerst nicht auffällt, dass im Programm Zeichenketten verschleiert werden. Im rechten Bereich ist der Assembler-Code der Funktion zur Prüfung der Seriennummer zu sehen. Hier fällt auf, dass kein verzweigter Kontrollfluss vorhanden ist, da keine Pfeile angezeigt werden. Die einzig sichtbare Funktionsaufrufe sind die der Trampolinfunktion. Der Angreifer hätte in der Praxis Schwierigkeiten, diese Funktion überhaupt zu finden, da sie gänzlich unverdächtig erscheint und der Funktionsaufruf, der zum Aufruf dieser Funktion führt, ebenfalls verschleiert ist.

- in der Funktion zur Ausführung des Lua-Programms (Zeile 47) wird die Funktion `MessageBox` zur Anzeige der Fehlermeldung bei unregistrierten Programmen genutzt (Zeile 54). Deshalb sollen alle Funktionsaufrufe verschleiert werden, damit der Angreifer nicht die Taktik nutzen kann, alle Aufrufe von `MessageBox` zu analysieren, um diese Stelle im Programm zu finden. Weiterhin soll die verdächtige Fehlermeldung durch eine unverdächtige Meldung getarnt werden (Zeile 53).

Das verschleierte Programm verhält sich zur Laufzeit identisch zur unverschleierten Variante. Auch die Übersetzung mit einem Compiler, der die Annotationen nicht versteht, führt zu einem funktionsfähigen Programm, das sich identisch verhält. Die statische Analyse im Rahmen eines Reverse Engineerings gestaltet sich allerdings nun schwieriger, da wesentliche Teile des Programms, die den Angreifern interessieren, verschleiert sind (vgl. Abbildung 25).

6.2 Benchmarks

Um die Auswirkung der einzelnen Verfahren auf Programmgröße und Laufzeitverhalten zu ermitteln, wurden verschiedene Benchmarks durchgeführt. Dazu wurde ein Computer mit folgender Spezifikation genutzt:

- Prozessor: Intel Core i7-2600 getaktet mit 3,4 GHz, 4 Kernen und Hyperthreading
- Hauptspeicher: 12 GB DDR3-Speicher getaktet mit 1.333 MHz
- Festplatte: Crucial M4, SSD, Kapazität 512GB
- Betriebssystem: Windows 7, 64 Bit

Sämtliche Durchschnittswerte in den folgenden Benchmarks beziehen sich jeweils auf den Median. Er wurde gegenüber dem arithmetischen Mittelwert bevorzugt, damit Ausreißer der Messwerte durch Hintergrundaufgaben des Betriebssystems nur geringe Auswirkungen auf das Ergebnis haben.

6.2.1 Kontrollflussverschleierung

Zur Untersuchung der Auswirkungen des modifizierten Verfahrens von Balachandran auf übersetzte Programme wurde zunächst das Dhrystone-Benchmark [Wei84] verwendet, da in diesem vielfältige Verzweigungen im Kontrollfluss auftreten. Der Quelltext des Benchmark-Programms wurde zunächst mit einer unmodifizierten Variante des TCC-Compilers (Version 0.9.26) übersetzt. Hierbei wurde bei 1.000 Läufen des Benchmarks eine durchschnittliche Punktzahl von 5.486 MIPS erzielt.

Um die Auswirkungen der Kontrollflussverschleierung auf die Laufzeitgeschwindigkeit des auf diese Weise verschleierten Programms zu messen, wurde der Compiler so modifiziert, dass für *jede* Funktion im Quelltext eine Kontrollflussverschleierung aktiviert wird. Nach Aktivierung dieser Verschleierung wurden im Benchmark nach 1.000 Läufen eine durchschnittliche Punktzahl von 512 MIPS erreicht, das Programm ist also grob um den Faktor 10 langsamer. Bei diesem Ergebnis ist allerdings zu bedenken, dass der Programmierer beim Einsatz der Verschleierung in der Praxis nur die Funktionen für eine Kontrollflussverschleierung annotieren sollte, die auch tatsächlich vor Reverse Engineering geschützt werden sollen. Wenn dies Funktionen zur Prüfung eines Kopierschutzes oder einer eingegebenen Seriennummer sind, wird der restliche Kontrollfluss des Programms in der Geschwindigkeit nicht beeinträchtigt. Dies ist ein Vorteil, der erst durch die Möglichkeit der gezielten Annotation ermöglicht wurde.

Als realistischer Anwendungsfall soll das Programm `bzip2` [Sew10] in der Version 1.0.6 verschleiert werden. In diesem wird unter anderem die Burrows-Wheeler-Transformation [Bur+94] zur Vorbereitung der Kompression genutzt. Die Funktionen dieses Algorithmus soll nun innerhalb von bzip2 verschleiert werden. Dies ist ein realistisches Szenario, analog zum Schutz eines nicht-öffentlichen Algorithmus innerhalb eines kommerziellen Programms. Dazu

wurden die entsprechenden Funktionen innerhalb von bzip2 annotiert, um deren Kontrollfluss zu verschleiern.

Zur Durchführung der Messung wurde eine Datei mit 100MB Zufallsdaten erzeugt, die anschließend von der unverschleierten und der verschleierten Variante von bzip2 komprimiert wurde. Bei 100 Läufen mit der unverschleierten Variante wurden zur Kompression der Datei durchschnittlich 47,24 Sekunden benötigt. Bei Verwendung des verschleierten Algorithmus stieg die durchschnittliche Zeit auf 57,59 Sekunden. Durch die Verschleierung eines einzelnen Algorithmus innerhalb des Programms ist die Ausführungszeit hier also nur um den Faktor 1,2 verlangsamt worden.

Die Programmgröße ist beim Dhrystone-Benchmark (Verschleierung aller Funktionen) von 12.288 bytes auf 13.824 Bytes angewachsen. Das Verfahren lässt die Größe der Programmdatei in diesem Fall also um grob 12% steigen. Im Falle des realistischeren bzip2-Benchmarks (Verschleierung der Burrows-Wheeler-Transformation) ist die Dateigröße von 116.737 Bytes auf 118.272 Bytes gestiegen, also nur grob 1%.

6.2.2 Opake Konstanten

Um die Auswirkung der Verschleierung von Funktionsaufrufen mithilfe der opaken Konstanten zu messen, wurde der Compiler zunächst so modifiziert, dass *alle* Funktionsaufrufe mit diesem Verfahren verschleiert werden. Da das oben benutze Dhrystone-Benchmark nicht viele Funktionsaufrufe benutzt, wurde stattdessen der Interpreter für die Skriptsprache Lua [Ier+95] in der Version 5.2.3 mit einer unmodifizierten TCC-Variante sowie mit der verschleiernden Variante übersetzt. Der Interpreter besitzt sehr viele Funktionsaufrufe, sodass er als Programm zum Testen dieser Verschleierung geeignet ist. Zum Quelltext von Lua gehört eine Test-Suite, die viele Funktionen des Interpreters testet. Diese wird benutzt, um möglichst viele Funktionsaufrufe im Interpreter zu provozieren.

Die Dauer des Tests beträgt bei 100 Läufen durchschnittlich 3,8 Sekunden pro Lauf, wenn der unverschleierte Interpreter verwendet wird. Bei der Verwendung des Interpreters, in dem alle Funktionsaufrufe verschleiert sind, wurde die Messung nach 30 Minuten abgebrochen, da der erste Lauf noch nicht abgeschlossen war. Dieser erhebliche Laufzeitunterschied ist allerdings auch leicht nachzuvollziehen – für *jeden* Funktionsaufruf müssen 32 SAT-Instanzen ausgewertet werden, um die Adresse des Aufrufs zu berechnen. Dieses Verschleierungsverfahren sollte also nur eingesetzt werden, wenn die verschleierten Funktionsaufrufe nicht in zeitkritischen Teilen des Programms geschehen.

Für eine praxisnäheres Szenario wurde wieder das bzip2-Programm analog zum vorigen Abschnitt verschleiert, sodass die Funktionsaufrufe der Burrows-Wheeler-Transformation verschleiert werden. Hier steigt die Ausführungszeit von 47,24 Sekunden auf 47,76 Sekunden. Die Ursache für diesen kaum messbaren Unterschied liegt darin begründet, dass die Funktionsaufrufe nur jeweils einmal pro 512 kB der zu verarbeitenden Datei ausgeführt werden müssen, insgesamt also nur rund 200 Funktionsaufrufe erfolgen.

Die Programmgröße des Lua-Interpreters (alle Funktionsaufrufe verschleiert) ist von 208.896 Bytes auf 3.365.376 Bytes gestiegen. Der erhebliche Zuwachs ist dadurch zu erklären, dass für *jeden* Funktionsaufruf eine SAT-Instanz mit Klauselmenge sowie Variablenbelegungen gespeichert werden muss. Beim bzip2-Test (nur Burrows-Wheeler-Transformation verschleiert) ist die Dateigröße von 116.737 Bytes auf 216.064 Bytes gewachsen, also um rund 85%.

Dieses Verfahren bietet zwar eine hohe Sicherheit, wenn sein Einsatz nicht bekannt ist (bei Bekanntsein könnte ein SAT-Solver die Konstanten schnell entschleiern), ist allerdings auch mit erheblichen Kosten verbunden. Durch den selektiven Einsatz mithilfe der Annotationen kann der Programmierer es allerdings gezielt an den Stellen einsetzen, bei denen API-Aufrufe des Betriebssystems oder ähnliches verschleiert werden sollen.

6.2.3 Zeichenkettenverschleierung

Die Auswirkung der Verschleierung von Zeichenketten auf die Laufzeit lässt sich nur schwierig in einem realistischen Rahmen testen – es gibt zwar einige Benchmarks und viele Programme, die Zeichenketten verarbeiten, aber verschleiert werden vom Verfahren nur *konstante* Zeichenketten. Deshalb wurde ein eigenes Benchmark entworfen, das eine konstante Zeichenkette an eine Funktion übergibt:

```
1   void doNothing(char *data) { }
2
3   void test() {
4       doNothing("This is a test string.");
5   }
6
7   int main() {
8       int i;
9       for(i = 0; i < 1000000000; i++) {
10          test();
11      }
12      return 0;
13  }
```

Das Benchmark ruft 1.000.000.000 Mal eine Funktion auf, die eine konstante Zeichenkette an eine andere Funktion übergibt, die sofort zurückkehrt. Optimierungen des Compilers können hierbei ausgeschlossen werden, da TCC über keine Optimierung verfügt, die in diesem Programm etwas optimieren könnte. Die Laufzeit beträgt bei 1.000 Läufen des Benchmarks durchschnittlich 4,5 Sekunden. Um die Verschleierung von Zeichenketten zu messen, wird die Zeichenkette im Programm mit einer Zeichenkette derselben Länge verschleiert. Die Laufzeit stieg dadurch auf durchschnittlich 24,2 Sekunden, was grob einer Verlangsamung um den Faktor 5 entspricht. Diese Verlangsamung ist für die Praxis allerdings nur von geringer Bedeutung, da ein Programm üblicherweise keine konstanten Zeichenketten verarbeitet. Es soll vielmehr an Stellen eingesetzt werden, an denen im Programm vorkommenden Zeichenketten den Angreifer zu einer für ihn relevanten Stelle führen würden.

Die Änderung der Dateigröße wird aus diesem Grund nicht gemessen, es lässt sich allerdings abschätzen, dass das Programm für jede verschleierte Zeichenkette um die Länge der Zeichen-

kette wächst, da zu dieser der gleichlange Schlüssel zur Entschleierung gespeichert werden muss.

6.2.4 Programmpunktverfahren

Um die Laufzeit der Verschleierung nach dem Programmpunktverfahren zu messen, müsste der Quelltext eines Programms so modifiziert werden, dass viele Programmpunkte darin annotiert sind und ebenfalls viele Funktionen zur Verschleierung mit dem korrekten Programmpunkt annotiert sind. Diese Annotationen können allerdings nicht automatisiert angebracht werden, da hierzu das Erreichbarkeitsproblem im Kontrollflussgraph gelöst werden müsste – aber genau auf dessen Unlösbarkeit basiert das Verfahren. Eine manuelle Annotation größerer Programme ist ebenfalls schwierig, da hierzu der genaue zeitliche Ablauf des Programms verstanden werden müsse.

Stattdessen hilft folgende Überlegung: Da jeder annotierte Programmpunkt für die korrekte Funktionsweise des Verfahrens nur einmal erreicht werden darf, ist der Einfluss auf die Laufzeit des Programms für die Aktualisierung der Programmpunkte *unabhängig* von der Komplexitätsklasse der tatsächlichen Programmlaufzeit. Relevant für die Laufzeit sind die mit den Programmpunkten durchgeführten Verschleierungen. Der Compiler wurde zum Testen daher so modifiziert, dass es nur einen Programmpunkt $(0, 1)$ beim Eintritt in die Funktion `main` gibt. Auf diese Weise können *alle* Funktionen des Programms den Programms so annotiert werden, dass sie mithilfe des Programmpunkts $(0, 1)$ verschleiert werden, denn dieser wurde genau einmal vorher durchlaufen. Durch diese Modifikation ist eine realistische Messung möglich.

Da das Verfahren Zugriffe auf Daten im Datensegment verschleiert, würde ein Test mit dem Dhrystone-Benchmark wenig Aussagekraft haben, da solche Zugriffe hier kaum genutzt werden. Besser geeignet ist der Lua-Interpreter, da hier viele globale Konstanten und Variablen genutzt werden. Die Laufzeit stieg hierbei von 3,8 Sekunden auf 3,9 Sekunden. Der geringe Unterschied ist dadurch zu erklären, dass für jeden verschleierten Zugriff nur eine einzige Maschinencode-Instruktion zusätzlich eingefügt werden muss. Die Programmgröße des Lua-Interpreters stieg von 208.896 Bytes auf 212.992 Bytes, was einem Zuwachs von grob 2% entspricht.

Bei dem im vorigen Abschnitt eingeführten Benchmark für Zeichenketten wurde das Verfahren ebenfalls getestet, da der Zugriff auf die Zeichenkette ebenfalls mit diesem Verfahren geschützt werden kann. Auch hier wurde kein signifikanter Unterschied in der Laufzeit gemessen, obwohl die Entschleierung hier sogar kontinuierlich in einer Schleife ausgeführt wird. Das Verfahren ist also trotz der Anwendung auf ein vollständiges Programm sehr schnell, benötigt wenig Platz und ist trotzdem sicherer als die anderen Verfahren.

Verfahren	Verlangsamung	Vergrößerung
Kontrollfluss	10	1,12
Zeichenketten	5	-
Opake Konstanten	?	16
Programmpunkte	1,02	1,02

Tabelle 26: Auswirkung der Verschleierung auf Programmgröße und Laufzeit im schlimmsten Fall: Es wurden jeweils Programme verschleiert, in denen die zu schützenden Fragmente sehr häufig vorkommen. Dabei wurde jeweils das gesamte Programm geschützt, sodass sich ein teils erheblicher Einfluss auf Programmgröße und -laufzeit zeigt. Der Einfluss des Verfahrens der opaken Konstanten auf die Laufzeit ist so groß, dass die Messung abgebrochen wurde.

Verfahren	Verlangsamung	Vergrößerung
Kontrollfluss	1,2	1,01
Zeichenketten	$< 1\%$	$< 1\%$
Opake Konstanten	1,01	1,85
Programmpunkte	$< 1\%$	$< 1\%$

Tabelle 27: Auswirkung der Verschleierung auf Programmgröße und Laufzeit im durchschnittlichen Fall: Hierzu wurden selektiv nur die Teile eines Programms geschützt, die aus Sicht des Programmierers tatsächlich gegen Reverse Engineering geschützt werden sollen. Abgesehen vom Verfahren der opaken Konstanten ist der Einfluss hierbei gering.

6.2.5 Fazit

Werden die vorgestellten Verfahren auf ein gesamtes Programm angewendet, so verschlechtert sich dessen Laufzeit teilweise erheblich (vgl. Tabelle 26). Der Vorteil des neuen Ansatzes, Verschleierung in den Compiler zu integrieren und nur selektiv zu durch den Programmieren zu aktivieren, ist es aber gerade, dass nicht das gesamte Programm verschleiert und damit verlangsamt werden muss. Aus diesem Grund sind die Daten dieses Benchmarks als „worst case“ zu interpretieren und werden bei der tatsächlichen Anwendung durch einen Programmierer nicht erreicht, sofern dieser die Annotationen so vornimmt, dass nicht die zeitkritischen Teile des Codes betroffen sind.

Bei der selektiven Anwendung auf sicherheitskritische Bereiche des Programms ist der Einfluss auf die Laufzeit und Codegröße gering (vgl. Tabelle 27). Durch die Benchmarks wurde gleichzeitig auch getestet, ob die Semantik der Programme durch Verschleierung verändert wird. Da sowohl die Testsuites von Lua als auch von bzip2 bei vollständig aktivierter Verschleierung erfolgreich durchlaufen worden, ist eine Korrektheit anzunehmen. Ein Beweis kann im Rahmen dieser Arbeit nicht erbracht werden, da schon der unmodifizierte TCC nicht auf semantische Korrektheit überprüft wurde.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurden zunächst bestehende Verfahren zur Verschleierung von Maschinenprogrammen untersucht. Dabei wurden verschiedene Nachteile festgestellt:

- Auswirkung auf Geschwindigkeit zur Laufzeit: Die untersuchten Verfahren verschleiern jeweils das gesamte Programm. Dadurch werden auch Teile des Programms verschleiert, die möglicherweise gar nicht schützenswert sind, durch den Schutz aber langsamer ausgeführt werden.
- Heimlichkeit: Das Verschleiern des gesamten Programms hat ebenfalls eine negative Auswirkung auf die Heimlichkeit des Verfahrens. Wenn beispielsweise der Kontrollfluss des Programms verschleiert wird und dadurch keine Sprünge mehr im disassemblierten Maschinencode vorkommen, so fällt einem Angreifer dies mithilfe von statistischen Analysen schnell auf. Er kann dadurch beliebige Stellen des Programms analysieren, um Spuren des Verfahrens zu finden und zu analysieren.
- Sicherheit: Einige der untersuchten Verfahren lassen sich durch partielle Auswertung mithilfe eines Disassemblers wie Kianxali automatisiert umkehren. Bei genauer Kenntnis des Verfahrens trifft dies auf alle untersuchten Verfahren zu – es kann also ein spezielles Skript zur Umkehrung des Verfahrens mithilfe von Kianxali entworfen werden.

Um diese Nachteile zu beseitigen, wurde zunächst ein neuer Ansatz zur Durchführung von Verschleierungen vorgestellt. Dazu wurde ein Compiler so modifiziert, dass Verschleierungstechniken direkt in den Compiler integriert werden können. Auf diese Weise können Verfahren so implementiert werden, dass bei der Umsetzung mehr über die Semantik des Quellprogramms bekannt ist, da alle Datentypen von beteiligten Operanden bekannt sind. Zur Evaluation dieses Ansatzes wurden einerseits bekannte Verfahren umgesetzt, um die Tauglichkeit des Ansatzes zu prüfen. Dabei wurde die Programmiersprache C um eine domänenspezifische Sprache zur Annotation von Verschleierungswünschen erweitert. Mithilfe dieser Annotationen kann der Programmierer genau kennzeichnen, an welcher Stelle der Compiler Verschleierung anwenden soll, sodass nun nicht mehr das gesamte Programm verschleiert wird.

Zusätzlich ermöglicht der neue Ansatz die Umsetzung von neuen Verfahren, die ohne die Integration in einen Compiler nur schwierig möglich wären. Bei der Ideenfindung für solche Verfahren wurde überlegt, auf welche Weise der Programmierer Wissen über das Programm beisteuern kann, das der Compiler zur Durchführung von Verschleierungen nutzen kann. Aus diesen Überlegungen ist das Programmpunktverfahren entstanden: Der Programmierer teilt dem Compiler über Annotationen mit, in welcher Reihenfolge bestimmte Punkte des Programms zur Laufzeit ausgeführt werden. Dieses Wissen ist nicht automatisiert berechenbar. Deshalb kann der Compiler die Informationen über die Programmpunkte nutzen, um damit Informationen des Programms zu verschleiern, ohne dass ein Angreifer dies automatisiert auf effiziente Weise umkehren kann.

Durch die Integration der Verfahren in einen Compiler sowie die Umsetzung des Programmpunktverfahrens ergaben sich folgende Vorteile:

- Steuerbare Auswirkung auf Geschwindigkeit zur Laufzeit: Da über die Annotationssprache nun selektiv gesteuert werden kann, welche Bereiche des Programms verschleiert werden kann, liegt die Auswirkung von Verschleierung auf die Geschwindigkeit des Programms nun in der Hand des Programmierers. Soll beispielsweise nur die Überprüfung einer Seriennummer oder die Implementierung eines anderen Kopierschutzes verschleiert werden, so wird sich die Verschleierung nicht auf das restliche Programm auswirken.
- Höhere Heimlichkeit: Durch die selektive Verschleierung des Programms fällt die Anwendung von Obfuscation im Programm insgesamt weniger auf. Ist beispielsweise der Kontrollflussgraph einer einzelnen Funktion verschleiert, so hat dies bei hinreichender Programmgröße kaum Einfluss auf statistische Tests, da in anderen Funktionen weiterhin Sprunganweisungen vorkommen. Der Angreifer muss die verschleierten Bereiche des Programms also zunächst identifizieren, bevor er die Verschleierung analysieren kann.
- Sicherheit: Wird vom Argument der höheren Heimlichkeit abgesehen, so ist die Sicherheit der bestehenden Verfahren durch die Integration in einen Compiler unverändert. Allerdings können neue Verfahren wie das vorgestellte Programmpunktverfahren eine höhere Sicherheit bieten, da sie über Annotationen Informationen des Programmierers erhalten können, die nicht effizient berechenbar sind. Auf diese Weise wird der Angreifer gezwungen, auf diese Weise geschützte Programme manuell oder ineffizient zu analysieren.

Ein Nachteil bei der Verwendung der Annotationssprache ist allerdings, dass der Programmierer nun über grundlegendes Wissen zum Thema Verschleierung verfügen muss. Während er bei bisherigen Verfahren einen Obfuscator verwenden konnte, der das gesamte Programm automatisiert verschleiert, muss er beim vorgestellten Ansatz Bereiche des Quellcodes seines Programms mit speziellen Annotationen versehen, die etwas Wissen über Verschleierung erfordern. Bei falscher Anwendung besteht die Gefahr, dass das Programm nicht hinreichend geschützt ist – und dies ist für den Programmierer möglicherweise nur schwer oder gar nicht zu prüfen, wenn er nur Anwender des Verfahrens ist. Beim Programmpunktverfahren wird sehr viel Verständnis über den Ablauf des Programms zur Laufzeit benötigt. Dieses Wissen ist in der Regel nur dem Programmierer selbst bekannt, bei nebenläufigen Programmen möglicherweise nur eingeschränkt. Dieses Verfahren ist also schwierig anzuwenden, wenn keine genauen Informationen über das Laufzeitverhalten des Programms bekannt sind, was ebenfalls ein Nachteil gegenüber bestehenden Verfahren ist.

7.2 Fazit

Die Integration von Obfuscation-Techniken in einen Compiler hat sich als zielführende Idee erwiesen. Die Implementierung von bestehenden Verfahren war problemlos möglich und konnte durch die selektive Anwendung einige Nachteile beseitigen, ohne dass die eigentlichen Verfahren geändert werden mussten. Ein weiterer Vorteil ist, dass der Programmierer nun

mithilfe von Annotationen Wissen über das Programm beisteuern kann, wodurch gänzlich neue Verschleierungstechniken denkbar sind. Die erwarteten Vorteile sind also eingetroffen.

Ein erwarteter Nachteil war, dass die Integration von Verschleierung in einen Compiler sehr viel aufwändiger ist, als textuelle Transformationen auf Assembler-Ebene im Rahmen eines Obfuscator-Skripts umzusetzen. Diese Vermutung konnte im Rahmen der Arbeit nicht vollständig untersucht werden, da durch die Wahl des Tiny C Compilers ein Übersetzer mit sehr geringer Komplexität gewählt wurde, sodass ein hoher Umsetzungsaufwand vermieden wurde.

7.3 Ausblick

Die Integration von Verschleierungstechniken in einen Übersetzer ermöglicht die Umsetzung von gänzlich neuen Verfahren zur Verschleierung, die ohne diese Integration nur schwierig möglich wären. Mit dem Programmpunktverfahren wurde bereits gezeigt, wie Wissen des Programmierers genutzt werden kann, um die Sicherheit eines Verfahrens zu stärken. Dieser Ansatz ist auch aus Sicht der theoretischen Informatik interessant, denn ein solches Verfahren läuft nicht vollständig automatisiert ab, sondern benutzt die Kreativität des Menschen in Form eines Orakels als weitere Eingabe. Dem Angreifer steht ein solches Orakel nicht zur Verfügung, wenn der Angriff automatisiert ablaufen soll. Hier könnten weitere Verfahren entwickelt werden, die Wissen des Programmierers über das Programm als Eingabe nutzen, um das Programm zu schützen.

Im Rahmen dieser Arbeit wurden die Verfahren in den Tiny C Compiler integriert, der aufgrund seiner One-Pass-Architektur keine Optimierung des Codes durchführt. Für eine praktische Nutzbarkeit der vorgestellten Verfahren müssten diese in einen optimierenden Compiler integriert werden, damit nicht bereits die Wahl des Compilers das Programm verlangsamt. Das Projekt *Obfuscator-LLVM* [Vau14] integriert Verschleierung in den Compiler LLVM, allerdings auf einer viel höheren Ebene: Obfuscator-LLVM führt Verschleierungen nicht auf dem Maschinencode durch, sondern auf der Zwischencode-Darstellung des Compilers. Auf diese Weise werden alle Frontends und Backends des Compilers unterstützt. Nachteilig ist allerdings, dass auf der Zwischencode-Darstellung viele der hier vorgestellten Verfahren nicht möglich wären, da sie mit der x86-Architektur und deren Maschinencode zusammenhängen. Auch die selektive Verschleierung mittels einer Annotationssprache wurde hier nicht umgesetzt, sondern wieder der gesamte Code verschleiert. Allerdings zeigt das Projekt die grundsätzliche Möglichkeit, Obfuscation auch in einen optimierenden Compiler zu integrieren. Es könnte somit einen Einstiegspunkt für weitere Arbeiten liefern.

A Auszüge des x86-Befehlssatzes

In diesem Abschnitt werden die Befehle der x86-Architektur erläutert, die in der Arbeit verwendet werden.

add a, b addiert b auf a

call a ruft ein Unterprogramm auf, indem der Instruktionszeiger auf den Stack gelegt und zur Adresse a gesprungen wird

cmp a, b vergleicht zwei Operanden, indem das Vorzeichen von $a - b$ berechnet und im Flag-Register gespeichert wird, wo es beispielsweise von einer bedingten Sprunganweisung ausgewertet werden kann

inc a erhöht a um eins

imul a, b multipliziert unter Berücksichtigung der Vorzeichen a mit b

jmp a unbedingter Sprung zur Adresse a

jz a jump if zero: springt zur Adresse a, falls das Zero-Flag gesetzt ist, beispielsweise bei einer cmp-Anweisung mit $a = b$

jnl a jump if not lower: springt zur Adresse a, falls Vorzeichen-Flag und Überlauf-Flag identisch sind, beispielsweise bei einer cmp-Anweisung mit $a \geq b$

jnz a jump if not zero: springt zur Adresse a, falls das Zero-Flag nicht gesetzt ist, beispielsweise bei einer cmp-Anweisung mit $a \neq b$

leave entfernt einen Prozedurrahmen vom Stack, indem die Adresse des Prozedurrahmens der aufrufenden Funktion wiederhergestellt wird

mov a, b kopiert den Inhalt von b nach a

movsx a, b kopiert den Inhalt von b nach a und behält dabei das Vorzeichen von b bei, auch falls b eine geringere Bitbreite als a hat

nop no operation: Anweisung ohne Effekt

push a legt den Operanden a als 32-Bit-Wert auf dem Stack ab

pushfd legt das Flag-Register auf dem Stack ab

popfd legt das oberste Element des Stacks im Flag-Register ab

retn a kehrt aus einem Unterprogramm zurück, indem zur auf dem Stack abgelegten Adresse gesprungen wird. Verkleinert dabei optional den Prozedurrahmen um a Byte, um den Platz der Prozedurparameter wieder freizugeben

rol a, b führt eine Ringschiebung der Bits von a um b Bitpositionen nach links durch

ror a, b führt eine Ringschiebung der Bits von a um b Bitpositionen nach rechts durch

sar a, b shift arithmetically right: Rechts-Shift von a um b Bit unter Beibehaltung des Vorzeichenbits

shr a, b shift right: Rechts-Shift von a um b Bit ohne Beibehalten des Vorzeichenbits

setz a kopiert den Wert des Zero-Flags nach a

sub a, b subtrahiert b von a

xor a, b führt eine exklusive Oder-Verknüpfung von a mit b durch

B Quellcode

B.1 Ungeschützte Variante der Beispielanwendung

Der Quellcode zeigt Auszüge aus dem Quellcode der Beispielanwendung, die im Rahmen der Arbeit zur Demonstration von Methoden des Angreifers angegriffen wird.

```
1  /* Checks whether a serial is valid, e.g. Folke: 4D581085 */
2  int checkSerial(char *name, char *given_serial_str) {
3      int nameLen = strlen(name), i;
4      unsigned int correct_serial = 0, given_serial;
5
6      if(nameLen == 0) {
7          return 0;
8      }
9
10     given_serial = strtoul(given_serial_str, NULL, 16);
11
12     for(i = 0; i < nameLen; i++) {
13         correct_serial ^= (name[i] ^ (0x90 + i)) * 0xABADF00D;
14     }
15     correct_serial ^= 0xCAFE;
16
17     return given_serial == correct_serial;
18 }
19
20
21 /* Dialog program for the registration dialog */
22 BOOL RegisterProc(HWND dlg, UINT msg, WPARAM wParam, LPARAM lParam) {
23     char name[20], serial[20];
24     switch(msg) {
25         case WM_COMMAND:
26             if(LOWORD(wParam) == IDOK) {
27                 GetDlgItemText(dlg, IDC_NAME, name, sizeof(name));
28                 GetDlgItemText(dlg, IDC_SERIAL, serial, sizeof(serial));
29                 if(checkSerial(name, serial)) {
30                     MessageBox(dlg, "Program registered",
31                         "Success", MB_ICONINFORMATION);
32                     EndDialog(dlg, 1);
33                 } else {
34                     MessageBox(dlg, "Invalid Serial!",
35                         "Error", MB_ICONEXCLAMATION);
36                 }
37             } else if(LOWORD(wParam) == IDCANCEL) {
38                 EndDialog(dlg, 0);
39             }
40     }
41     return FALSE;
42 }
43
44
45 /* Called when the user clicks File -> Run */
46 void onMenuRun(HWND hwnd, HWND lua_text_win, int registered) {
47     int len;
```

```
48      char *lua_code;
49
50      len = GetWindowTextLength(lua_text_win) + 1;
51      if(!registered && len > 105) {
52          MessageBox(hwnd, "This demo version only supports short programs",
53          "Error", MB_ICONEXCLAMATION);
54          return;
55      }
56      lua_code = (char *) malloc(sizeof(char) * len);
57      if(lua_code == NULL) {
58          MessageBox(hwnd, "Out of memory", "Error", MB_ICONERROR);
59          return;
60      }
61      GetWindowText(lua_text_win, lua_code, len);
62      lua_exec(hwnd, lua_code);
63      free(lua_code);
64  }
65
66
67
68
69  /* Windows calls this function whenever something happens to our window */
70  LRESULT WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
71      /* Stores the window handle of the lua code edit control */
72      static HWND lua_text;
73
74      /* Stores whether the program is registered */
75      static int registered = 0;
76
77      switch (message) {
78      case WM_CREATE:
79          /* Sent once upon creation of the main window: Create controls */
80          SetMenu(hwnd, LoadMenu(NULL, MAKEINTRESOURCE(IDR_MENU)));
81          lua_text = createTextArea(hwnd, 0, 0, 0, 1, 1);
82          SetWindowText(lua_text, LUA_PROG);
83          cloneWindowSize(hwnd, lua_text);
84          SetFocus(lua_text);
85          break;
86      case WM_SIZE:
87          /* Sent when the window is resized: Resize lua control as well */
88          cloneWindowSize(hwnd, lua_text);
89          break;
90      case WM_COMMAND:
91          /* Sent when the user uses a control */
92          switch(LOWORD(wParam)) {
93              case ID_HELP_REGISTER:
94                  /* Help->Register clicked, process registration dialog */
95                  if(DialogBox(NULL, MAKEINTRESOURCE(IDD_REGISTER),
96                      hwnd, (DLGPROC) RegisterProc))#
97                  {
98                      /* successfully registered */
99                      registered = 1;
100                     EnableMenuItem(GetMenu(hwnd),
101                         ID_HELP_REGISTER, MF_DISABLED);
102                 }
```

```
103                 break;
104             case ID_FILE_RUN:
105                 /* File -> Run clicked, execute lua code */
106                 onMenuRun(hwnd, lua_text, registered);
107                 break;
108         }
109         break;
110     case WM_DESTROY:
111         /* Called when the user closes the window: leave the msg loop */
112         PostQuitMessage(0);
113         break;
114     default:
115         return DefWindowProc(hwnd, message, wParam, lParam);
116     }
117     return 0;
118 }
119
120
121 /* Entry point of the program: Creates main window */
122 int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
123     LPSTR lpCmdLine, int nCmdShow)
124 {
125     MSG msg;
126     WNDCLASSEX wc = {0};
127     HWND hwnd;
128     const char *className = "Crackme";
129
130     /* Get screen resolution to center window */
131     int deskWidth = GetSystemMetrics(SM_CXSCREEN);
132     int deskHeight = GetSystemMetrics(SM_CYSCREEN);
133
134     /* Setup window creation */
135     wc.cbSize = sizeof(wc);
136     wc.hInstance = hInstance;
137     wc.lpszClassName = className;
138     wc.lpfnWndProc = (WNDPROC) WndProc;
139     wc.hbrBackground = (HBRUSH) GetStockObject(LTGRAY_BRUSH);
140     wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
141     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
142
143     if(!RegisterClassEx(&wc)) {
144         MessageBox(0, "Couldn't register window class",
145             "Error", MB_ICONERROR);
146         return -1;
147     }
148
149     /* Create main application window, also sends WM_CREATE to WndProc */
150     hwnd = CreateWindowEx(0,
151             className,
152             "CrackMe",
153             WS_OVERLAPPEDWINDOW,
154             deskWidth / 4, deskHeight / 4, 640, 480,
155             0, 0, hInstance, 0);
156
157     if(hwnd == NULL) {
```

```
158         MessageBox(0, "Couldn't create window", "Error", MB_ICONERROR);
159         return -1;
160     }
161
162     ShowWindow(hwnd, nCmdShow);
163     UpdateWindow(hwnd);
164
165     /* Main application loop: Wait for messages and forward them to
166      * WndProc. The loop ends when PostQuitMessage is called.
167      * The argument to PostQuitMessage is stored in msg.wParam
168      * and returned as exit code.
169      */
170     while(GetMessage(&msg, NULL, 0, 0) > 0) {
171         TranslateMessage(&msg);
172         DispatchMessage(&msg);
173     }
174     return msg.wParam;
175 }
```

B.2 Geschützte Variante der Beispielanwendung

In dieser Variante wurden Annotationen in den Quelltext eingefügt, die die Semantik des Programms nicht verändern, bei der Verwendung des modifizierten Compilers aber den Schutz vor Reverse-Engineering-Angriffen erhöhen.

```
1  /* Checks whether a serial is valid, e.g. Folke: 4D581085 */
2  #pragma obfuscate_function(control_flow, function_calls)
3  int checkSerial(char *name, char *given_serial_str) {
4      int nameLen = strlen(name), i;
5      unsigned int correct_serial = 0, given_serial;
6
7      if(nameLen == 0) {
8          return 0;
9      }
10
11     given_serial = strtoul(given_serial_str, NULL, 16);
12
13     for(i = 0; i < nameLen; i++) {
14         correct_serial ^= (name[i] ^ (0x90 + i)) * 0xABADF00D;
15     }
16     correct_serial ^= 0xCAFE;
17
18     return given_serial == correct_serial;
19 }
20
21 /* Dialog program for the registration dialog */
22 #pragma obfuscate_function(data_access(0, 4))
23 BOOL RegisterProc(HWND dlg, UINT msg, WPARAM wParam, LPARAM lParam) {
24     char name[20], serial[20];
25     switch(msg) {
26         case WM_COMMAND:
27             if(LOWORD(wParam) == IDOK) {
28                 GetDlgItemText(dlg, IDC_NAME, name, sizeof(name));
29                 GetDlgItemText(dlg, IDC_SERIAL, serial, sizeof(serial));
30                 if(checkSerial(name, serial)) {
31                     MessageBox(dlg, "Program registered",
32                         "Success", MB_ICONINFORMATION);
33                     EndDialog(dlg, 1);
34                 } else {
35                     MessageBox(dlg, "Invalid Serial!",
36                         "Error", MB_ICONEXCLAMATION);
37                 }
38             } else if(LOWORD(wParam) == IDCANCEL) {
39                 EndDialog(dlg, 0);
40             }
41     }
42     return FALSE;
43 }
44
45 /* Called when the user clicks File -> Run */
46 #pragma obfuscate_function(function_calls)
47 void onMenuRun(HWND hwnd, HWND lua_text_win, int registered) {
48     int len;
49     char *lua_code;
```

```
50
51      len = GetWindowTextLength(lua_text_win) + 1;
52      if(!registered && len > 105) {
53  #       pragma obfuscate_string("State: Ok", "Debug")
54          MessageBox(hwnd,
55              "This demo version only supports short programs",
56              "Error", MB_ICONEXCLAMATION);
57          return;
58      }
59      lua_code = (char *) malloc(sizeof(char) * len);
60      if(lua_code == NULL) {
61          MessageBox(hwnd, "Out of memory", "Error", MB_ICONERROR);
62          return;
63      }
64      GetWindowText(lua_text_win, lua_code, len);
65      lua_exec(hwnd, lua_code);
66      free(lua_code);
67  }
68
69  /* Windows calls this function whenever something happens to our window */
70  LRESULT WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
71      /* Stores the window handle of the lua code edit control */
72      static HWND lua_text;
73
74      /* Stores whether the program is registered */
75      static int registered = 0;
76
77      switch (message) {
78      case WM_CREATE:
79          /* Sent once upon creation of the main window: Create controls */
80  #       pragma obfuscation_checkpoint(0, 3)
81          SetMenu(hwnd, LoadMenu(NULL, MAKEINTRESOURCE(IDR_MENU)));
82          lua_text = createTextArea(hwnd, 0, 0, 0, 1, 1);
83          SetWindowText(lua_text, LUA_PROG);
84          cloneWindowSize(hwnd, lua_text);
85          SetFocus(lua_text);
86          break;
87      case WM_SIZE:
88          /* Sent when the window is resized: Resize lua control as well */
89          cloneWindowSize(hwnd, lua_text);
90          break;
91      case WM_COMMAND:
92          /* Sent when the user uses a control */
93          switch(LOWORD(wParam)) {
94              case ID_HELP_REGISTER:
95                  /* Help->Register clicked, process registration dialog */
96                  if(DialogBox(NULL, MAKEINTRESOURCE(IDD_REGISTER),
97                      hwnd, (DLGPROC) RegisterProc))#
98                  {
99                      /* successfully registered */
100                     registered = 1;
101                     EnableMenuItem(GetMenu(hwnd),
102                         ID_HELP_REGISTER, MF_DISABLED);
103                 }
104                 break;
```

```
105             case ID_FILE_RUN:
106                 /* File -> Run clicked, execute lua code */
107                 onMenuRun(hwnd, lua_text, registered);
108                 break;
109         }
110         break;
111     case WM_DESTROY:
112         /* Called when the user closes the window: leave the msg loop */
113         PostQuitMessage(0);
114         break;
115     default:
116         return DefWindowProc(hwnd, message, wParam, lParam);
117     }
118     return 0;
119 }
120
121 /* Entry point of the program: Creates main window */
122 int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
123     LPSTR lpCmdLine, int nCmdShow)
124 {
125     MSG msg;
126     WNDCLASSEX wc = {0};
127     HWND hwnd;
128     const char *className = "Crackme";
129
130     /* Get screen resolution to center window */
131     int deskWidth = GetSystemMetrics(SM_CXSCREEN);
132     int deskHeight = GetSystemMetrics(SM_CYSCREEN);
133
134 #   pragma obfuscation_checkpoint(0, 1)
135     /* Setup window creation */
136     wc.cbSize = sizeof(wc);
137     wc.hInstance = hInstance;
138     wc.lpszClassName = className;
139     wc.lpfnWndProc = (WNDPROC) WndProc;
140     wc.hbrBackground = (HBRUSH) GetStockObject(LTGRAY_BRUSH);
141     wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
142     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
143
144     if(!RegisterClassEx(&wc)) {
145         MessageBox(0, "Couldn't register window class",
146             "Error", MB_ICONERROR);
147         return -1;
148     }
149 #   pragma obfuscation_checkpoint(0, 2)
150
151     /* Create main application window, also sends WM_CREATE to WndProc */
152     hwnd = CreateWindowEx(0,
153             className,
154             "CrackMe",
155             WS_OVERLAPPEDWINDOW,
156             deskWidth / 4, deskHeight / 4, 640, 480,
157             0, 0, hInstance, 0);
158
159     if(hwnd == NULL) {
```

```
160         MessageBox(0, "Couldn't create window", "Error", MB_ICONERROR);
161         return -1;
162     }
163
164     ShowWindow(hwnd, nCmdShow);
165     UpdateWindow(hwnd);
166
167 #   pragma obfuscation_checkpoint(0, 4)
168
169     /* Main application loop: Wait for messages and forward them to
170      * WndProc. The loop ends when PostQuitMessage is called.
171      * The argument to PostQuitMessage is stored in msg.wParam
172      * and returned as exit code.
173      */
174     while(GetMessage(&msg, NULL, 0, 0) > 0) {
175         TranslateMessage(&msg);
176         DispatchMessage(&msg);
177     }
178
179 #   pragma obfuscation_checkpoint(0, 5)
180
181     return msg.wParam;
182 }
```

B.3 Kianxali-Script zur Analyse selbstmodifizierenden Codes

Dieses Skript kann im Disassembler Kianxli [Wil14] verwendet werden, um selbstmodifzierenden Code aufzudecken und automatisiert auszuwerten. Dies ist insbesondere bei der automatisierten Umkehrung des Verfahrens von Balachandran [BE13] sinnvoll.

```
# Iterates all instructions and displays write-access to code,
# i.e. finds self-modifying code.
# Also tries to apply the changes statically to aid further analyzation.

# Changed addresses will be remembered here in order to reanalyze them later
changes = []

$api.traverseCode do |inst|
  for op in inst.getDestOperands
    # Check if operand has code address as destination
    destAddr = op.asNumber
    next unless $api.isCodeAddress(destAddr)

    # Try patching by evaluating the modifying instruction
    instAddr = inst.getMemAddress
    destBits = op.getPointerDestSize()
    original = $api.readBits(destAddr, destBits)
    case inst.getMnemonic
    when "XOR"
      puts "#{instAddr.to_s(16)} modifies code at #{destAddr.to_s(16)} (XOR)"
      $api.patchBits(destAddr,
          original ^ inst.getSrcOperands.first.asNumber, destBits)
      changes << destAddr
    when "ADD"
      puts "#{instAddr.to_s(16)} modifies code at #{destAddr.to_s(16)} (ADD)"
      $api.patchBits(destAddr,
          original + inst.getSrcOperands.first.asNumber, destBits)
      changes << destAddr
    else
      puts "Unhandled mnemonic at #{instAddr.to_s(16)}: #{inst.getMnemonic}"
    end
  end
end

# Now reanalyze the changed addresses
changes.each {|dst| $api.reanalyze(dst)}

puts "#{changes.size} self-modifying instructions patched"
```

C Inhalt des git-Repositories

Dieser Abschnitt erläutert den Aufbau des git-Repositories `2014-fwi-ma`, in dem die praktischen Ergebnisse dieser Arbeit abgelegt sind.

- thesis: enthält den LaTeX-Code dieses Dokuments
- tinycc: enthält den Quelltext des modifizierten Tiny C Compilers. Neu sind insbesondere die Dateien `sat.c`, `sat.h`, `obfuscation.c`, `obfuscation.h` sowie `lib/obflib.c`. Die letzte Datei enthält Funktionen, die zur Umsetzung der Verfahren in das Zielprogramm gelinkt werden.
- tests: enthält das Beispielprogramm sowie die durchgeführten Benchmarks:
  - bzip2: Eine mit TCC kompilierbare Variante des Kompressionsprogramms `bzip2`. Angepasst wurde nur das Makefile, um Unterstützung für TCC hinzuzufügen.
  - crackme: Der Code der Beispielanwendung inklusive der Annotationen zum Schutz des Programms. Das beiliegende Makefile kann zum Kompilieren der Anwendung mit der TCC-Variante aus dem übergeordneten Verzeichnis genutzt werden.
  - dhrystone: Ein an TCC angepasstes Dhrystone-Benchmark inklusive Makefile.
  - lua: Eine mit TCC kompilierbare Variante des Lua-Interpreters. Angepasst wurde nur das Makefile. Enthält ebenfalls die Lua-Testsuite, mit der die Korrektheit des Interpreters getestet werden kann.
- presentation: enthält den LaTeX-Code der Präsentation dieser Masterarbeit

D Literatur

[Arb02] Geneviève Arboit.
„A Method for Watermarking Java Programs via Opaque Predicates“.
In: *In Proc. Int. Conf. Electronic Commerce Research (ICECR-5*. 2002. URL:
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.6545.

[Bar+12] Boaz Barak u. a. „On the (Im)Possibility of Obfuscating Programs“.
In: *J. ACM* 59.2 (Mai 2012), 6:1–6:48. ISSN: 0004-5411.
DOI: 10.1145/2160158.2160159.

[BE13] V. Balachandran und S. Emmanuel. „Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code“. In: *IEEE Transactions on Information Forensics and Security* 8.4 (2013), S. 669–681. ISSN: 1556-6013.
DOI: 10.1109/TIFS.2013.2250964.

[Bel13] Fabrice Bellard. *Tiny C Compiler*. Feb. 2013.
URL: http://www.tinycc.org (besucht am 20. 07. 2014).

[BS05] Arini Balakrishnan und Chloe Schulze. *Code Obfuscation Literature Survey*. 2005.
URL:
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.7043.

[Bur+94] M. Burrows u. a. *A block-sorting lossless data compression algorithm*. Techn. Ber. Systems Research Center, 1994. URL:
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.6177.

[CT02] Christian S. Collberg und Clark D. Thomborson. „Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection.“
In: *IEEE Transactions on Software Engineering* 28.8 (2002), S. 735–746.
DOI: 10.1109/TSE.2002.1027797.

[GM12] Roberto Giacobazzi und Isabella Mastroeni. „Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation“. English. In: *Static Analysis*. Hrsg. von Antoine Miné und David Schmidt. Bd. 7460.
Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 129–145.
ISBN: 978-3-642-33124-4. DOI: 10.1007/978-3-642-33125-1_11.

[HR] Hex-Rays. *IDA Pro*.
URL: https://www.hex-rays.com/products/ida/index.shtml (besucht am 09. 08. 2014).

[Ier+95] Roberto Ierusalimschy u. a. „Lua – an Extensible Extension Language“. In: *Software: Practice and Experience* 26 (1995), S. 635–652. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.6432.

[Int14] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Basic Architecture*. Bd. 1. Intel, 2014. URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[IS14] The On-Line Encyclopedia of Integer Sequences. *A000522*. 2014. URL: https://oeis.org/A000522 (besucht am 20. 07. 2014).

[Kin10] Johannes Kinder. „Static Analysis of x86 Executables“. Diss. Technische Universität Darmstadt, Nov. 2010. URL: http://tuprints.ulb.tu-darmstadt.de/2338/.

[KPW12] Markus Kammerstetter, Christian Platzer und Gilbert Wondracek. „Vanity, Cracks and Malware: Insights into the Anti-copy Protection Ecosystem“. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. New York, NY, USA: ACM, 2012, S. 809–820. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382282.

[Mic] Microsoft. *Microsoft Developer Network*. URL: http://msdn.microsoft.com/en-us/windows/desktop/ (besucht am 20. 07. 2014).

[MKK07] A. Moser, C. Kruegel und E. Kirda. „Limits of Static Analysis for Malware Detection“. In: *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. Dez. 2007, S. 421–430. DOI: 10.1109/ACSAC.2007.21.

[Sew10] Julian Seward. *bzip2*. 2010. URL: http://www.bzip.org/ (besucht am 01. 09. 2014).

[SH12] Michael Sikorski und Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. San Francisco, CA, USA: No Starch Press, 2012. ISBN: 9781593272906.

[SML96] Bart Selman, David G. Mitchell und Hector J. Levesque. „Generating Hard Satisfiability Problems“. In: *Artificial Intelligence* 81.1–2 (1996). Frontiers in Problem Solving: Phase Transitions and Complexity, S. 17 –29. ISSN: 0004-3702. DOI: 10.1016/0004-3702(95)00045-3.

[Vau14] Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud. *Obfuscator-LLVM*. Aug. 2014. URL: http://o-llvm.org/ (besucht am 01. 08. 2014).

[War02] Henry S. Warren. *Hacker's Delight*.
Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
ISBN: 0201914654.

[Wei84] Reinhold P. Weicker.
„Dhrystone: A Synthetic Systems Programming Benchmark".
In: *Commun. ACM* 27.10 (Okt. 1984), S. 1013–1030. ISSN: 0001-0782.
DOI: 10.1145/358274.358283.

[Wil14] Folke Will. „Kianxali: Kieler Analyzer for Executables and Libraries".
Master-Projekt. Christian-Albrecht-Universität zu Kiel, 2014.
URL: http://code.google.com/p/kianxali/.

[Xu+05] Ke Xu u. a. „A Simple Model to Generate Hard Satisfiable Instances".
In: *CoRR* abs/cs/0509032 (2005). URL:
http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.384.

[Yus13] Oleh Yuschuk. *OllyDbg*. 2013.
URL: http://www.ollydbg.de/ (besucht am 20. 07. 2014).