SAFE: Eine verifizierte Informationsflussarchitektur

Denis Küchemann
Seminar Theorie der Programmierung
Softwareentwicklung und Verifikation
Institut für Informatik

14. Juni 2015

Zusammenfassung

Diese Seminararbeit befasst sich mit dem formalen Aufbau der SAFE-Hochsicherheitsarchitektur. Es wird die Unterteilung in konkrete Maschine mit Cache-Fault-Handler und dem abstrakten Gegenpart, sowie die Auslagerung der Informationsflusskontrolle in eine symbolische Maschine analysiert. Für eine Verifizierung der gegenseitigen Verfeinerung der Maschinen inkl. Cache-Fault-Handler sowie Nichtinterferenz werden die Kernkomponenten von SAFE formalisiert und Beweisstrategien skizziert.

1 Einführung

SAFE (**S**emantically **A**ware **F**oundation **E**nvironment) ist eine Hochsicherheitsarchitektur für die Kontrolle von Informationsflüssen (IFC, engl.: information flow control), die das Legacy-Design bisheriger Systeme und Chips aufgibt. Die überschüssige Rechenleistung moderner Systeme soll genutzt werden, um jegliche Daten in Speicher und Registern bis hin zum 16-Bit Wort mit Tags/Labels (Wort-Größe) anhand von Vertraulichkeitsrichtlinien zu versehen, statt übliche Software-Lösungsansätze zu benutzen. Die dynamische Überprüfung via Software ist im Vergleich zur Überprüfung via Hardware zu teuer; die Benutzung von Hardware ermöglicht darüber hinaus die Abdeckung der häufigsten Angriffsszenarien wie Skriptattacken/Maschinen-Code-Injection. [1]
Daten können z.B. so gekennzeichnet werden, dass sie nur von einer bestimmten Entität (engl.: principal), z.B. einer Person, genutzt werden können oder noch nicht für eine Weiterverarbeitung bereit sind. Zu diesem Zweck befindet sich im Prozessor des SAFE-Modells die Tag-Management-Unit (TMU), die die Tagging-Mechanismen hardwaremäßig einbindet. [1]
SAFE unterstützt das dynamische Monitoring sensibler Daten (Tracking) von sowohl impliziten wie auch expliziten Informationsflüssen für beliebigen Code, statt lediglich für Code, der bereits durch statische Code-Analyse akzeptiert wurde.
Diese Seminararbeit beschäftigt sich insbesondere mit dem von Pierce et al. verfassten Paper A Verified Information-Flow Architecture [2].

In Kapitel 2 wird zuerst ein Überblick über die generelle Funktionsweise von SAFE geschaffen. Kapitel 3 erläutert einige Features von SAFE, die für eine vereinfachte Modellierung eingeschränkt werden. Anschließend werden in den Kapiteln 4 und 5 die Aufteilung in abstrakte, symbolische und konkrete Maschine sowie der Cache-Fault-Handler eingeführt. Kapitel 6 skizziert eine Verifizierung des Cache-Fault-Handlers. In Kapitel 7 wird die Verfeinerung zwischen den Maschinen formalisiert, um in Kapitel 8 die Nichtinterferenz zu zeigen. Abschließend wird im Kapitel Resümee und Ausblick zusammengefasst, welche Themen behandelt wurden und ein Ausblick gegeben.

2 SAFE-Funktionsweise

2.1 Tagging

Ein Tag ist ein beliebiger Zeiger [3], der Informationen über die Rechte einer Entität enthält. Das Tag könnte z.B. auf eine Struktur zeigen, die erfordert, dass für einen Benutzer das Recht installiert ist, auf einen Wert zuzugreifen. [4] Das Tagging („Tag-Verbreitung“, engl.: tag propagation) geschieht mittels der Hardware. Wie ein Tag propagiert und interpretiert wird, entscheidet die Software.
Die Tag-Propagation ist eine partielle Funktion von Argument-Tupeln (*opcode*, *pc tag*, *argument*1 *tag*, *argument*2 *tag*, ...) auf Ergebnis-Tupel (*new pc tag*, *result tag*). Argument-Tupel werden bei Instruktionsaufrufen erstellt; Ergebnis-Tupel werden erstellt, falls der Instruktionsaufruf erfolgreich war. Neben dem *opcode*, der eine Maschineninstruktion darstellt (Integerwert), enthält das Argumenttupel das *pctag*, das für das Tag des Befehlszählers der CPU (PC, engl.: program counter) steht. Ebenso sind die Tags der erwarteten Argumente des Opcodes enthalten.
Das Ergebnistupel enthält ein Tag für den neuen PC (*new pc tag*) und ein Tag für neu erstelle Daten (*result tag*), sofern durch die Instruktion neue Daten erstellt werden.

Instruktionsaufruf Die Ausführung einer Instruktion ist erlaubt, falls die nächste auszuführende Instruktion *opcode* ist, der derzeitige PC mit *pctag* getaggt ist und die erwarteten Argumente mit *argument*1 *tag*, *argument*2 *tag*, ... getaggt sind. Das entstehende Ergebnistupel repräsentiert den neuen Maschinenzustand.

2.2 Regel-Caching & Cache-Fault-Handler

Die Argumenttupel/Ergebnistupel-Paare werden Regelinstanzen (*rule instances*) genannt. Kürzlich benutzte Regelinstanzen werden im Hardware-Regel-Cache (*rule cache*) behalten. Tritt ein Cache-Miss auf, wird der *Cache Fault Handler* aufgerufen (siehe Kapitel 5).
Der Cache-Fault-Handler prüft, ob eine Verletzung der Richtlinien vorliegt, oder ob die Tag-Kombination erlaubt und die Regelinstanz zum Regel-Cache hinzugefügt werden kann.

3 SAFE-Features

3.1 Software

Das in Assembler und Tempest (System-Programmiersprache) programmierte SAFE bietet neben Prozess-Scheduling, streambasierte Interprozesskommunikation und Speicherverwaltung/Garbage-Collection. Die SAFE-Dienste sollen sich dabei gegenseitig so überwachen, dass ein Angreifer immer erst mehrere verschiedene Sicherheitsbereiche umgehen muss, um einen vollständigen Zugriff auf die Maschine zu erlangen. Dieses Ziel soll durch die Installation eines Zero-Kernels erreicht werden, der keinen omni-privilegierten Code enthält. [5]

3.2 Hardware

Der Fokus von SAFE liegt auf dem Management generischer und dynamisch typisierter Low-Level-Tagging-Hardware. Datenworte sind alle fix z.B. als Integerwert oder Zeiger markiert. SAFE bietet Speichersicherheit; die Zeiger (Fat-Pointers) bestehen aus einem 64Bit-Tripel ($Basis, Grenzen, Offset$). Zeigeroperationen beinhalten eine Prüfung dieser Grenzen. Speicher- und Registerworte sowie der PC werden mit einem 59Bit-Tag versehen. Rich-Tag-Daten können folgendermaßen dargestellt werden [6] (je nach Quelle befindet sich die atomistische Gruppe vor dem Tag):

59 Bit	5 Bit	64 Bit
Tag	atomistische Gruppe	Payload
von Software interpretiert	Int64, Zeiger, Double, Instruktion, ...	Daten

Der Regel-Cache ist durch eine Kombination verschiedener Hash-Funktionen implementiert, um einen vollassoziativen Cache zu approximieren.

3.3 Formale Modellierung & Verifizierung

Ein weiterer Fokus von SAFE liegt in der Formalisierung jeglicher SAFE-Funktionalität. Die Kern-Features konnten bereits auf Breeze-Level[1] und auch auf abstraktem sowie Hardware-Level formal beschrieben werden.

Vereinfachung des Modells Um die Arbeit mit dem Modell zu vereinfachen, werden einige Features von SAFE nicht beachtet.
Zwar sind die Tagging-Mechanismen generisch; diese Arbeit analysiert allerdings eine simplifizierte IFC und Nichtinterferenz[2]. Breeze und Tempest für die fokussierte Hardware- und Laufzeitdienst-Analyse werden ignoriert. Der Instruktionssatz wird auf wenige Opcodes begrenzt (siehe Appendix A) und ein Stack statt Registern verwendet. Statt der feinkörnigen Privilegienunterscheidung von SAFE wird die bekannte Aufteilung zwischen Benutzer- und Kernel-Modus benutzt. Im Regel-Cache kann sich lediglich ein Eintrag (im Kernel-Speicher) befinden. Die Analyse von Verdrängungs- und Ersetzungsstrategien oder Cache-Hardware ist somit unnötig. Eine Richtlinienverletzung wird lediglich mit einem Maschinenstopp behandelt. Eine immense Modellvereinfachung wird durch die Benutzung eines einzelnen deterministischen Threads erreicht.

4 Aufteilung in abstrakte, symbolische und konkrete IFC-Maschine

Die abstrakte Maschine verkörpert die IFC-Mechanismen, die durch die konkrete Maschine implementiert werden und ist damit das Kern-Modell von SAFE. Anhand der abstrakten Maschine sollen die Funktionalität von SAFE und Nichtinterferenz verifiziert werden. Dabei ist Determinismus eine gewünschte Eigenschaft, da das Beweisen sonst weiter verkompliziert wird [6]. Durch Gegebenheit von Determinismus bleibt die Nichtinterferenz-Eigenschaft der abstrakten Maschine in der konkreten Maschine erhalten. Mit Coq[3] wurden die Kern-Funktionen der IFC-Unterstützung modelliert und eine Beweisstrategie entwickelt. Dennoch ist es schwierig und teils problematisch, Maschinencode zu verifizieren, wenn man die Outputs der abstrakten und der konkreten Maschine auf Kompatibilität (Track-Matching) prüft. Durch Hinzufügen der symbolischen Regel-Maschine, die als Brücke zwischen abstrakter und konkreter Maschine agiert, ist es möglich, die IFC-Regeln auszulagern. Die symbolische ist der abstrakten Maschine ähnlich, jedoch ist die Definition der Schrittrelation (siehe Kapitel 4.1) verschieden, wobei Maschinenzustände und Verhalten der Schrittrelation identisch sind.

4.1 Abstrakte Maschine und Einführung der formalen Notation

Atome und Label-Menge Integerwerte werden durch n, m und p dargestellt. Vorerst werden Zeiger und Integerwerte nicht unterschieden. In der Label-Menge $\mathcal{L}$ befinden sich die IFC-Label L, die die Integerwerte schützen sollen. Die Label-Menge ist augmentiert mit der Halbordnung $\leq$, der Supremumsoperation $\vee$ und einem Bottom-Element $\bot$ mit $\bot \leq \top$ und $\bot \vee \top = \top$, wobei $\top$ das Top-Element ist. Öffentliche Daten werden mit $\bot$ („low“) gelabelt, während private Daten mit $\top$ („high“) gelabelt werden. Aus einer Verbindung (Join - $\vee$) von zwei Variablen verschiedener Sicherheitsstufen entsteht eine Variable mit der höheren der beiden Sicherheitsstufen. [7]
Ein durch L geschützter Integerwert n wird als Atom $a = n@L$ bezeichnet. Im Initialzustand der Maschine sind bereits sämtliche Daten mit Labels versehen.

[1] Breeze ist eine funktionale Applikationsprogrammiersprache mit Sicherheitsfokus. Sie wird für die Programmierung im Benutzermodus angewendet.

[2] Nichtinterferenz (engl.: non-interference) bedeutet, dass geheime Informationen die öffentliche Dateneinsicht nicht verändern.

[3] Coq ist ein Programm, dass maschinengestütztes Beweisen ermöglicht.

Speicher Der Stack σ ist eine Liste von Atomen und unterteilt sich in Stacks für reguläre Atome (a, σ) und Stacks, die mit Rücksprungadressen beginnen $(pc; \sigma)$. Der pc bezeichnet den Befehlszähler (engl.: program counter). SAFE verfügt über einen Instruktionsspeicher ι und einen Datenspeicher μ. Die partiellen Funktionen ι und μ bilden von Adressen auf Instruktionen respektive Atome ab. $\mu(p) \leftarrow a = \mu'$ bedeutet, dass der Speicher μ' mit μ übereinstimmt außer an der Stelle p, wo der Wert a ist.

Abstrakter Maschinenzustand und Schrittrelation Ein abstrakter Maschinenzustand wird mit $\langle \mu\ [\sigma]\ pc \rangle$ bzw. $\mu\ [\sigma]\ pc$ angegeben. Das pc-Label wird für das Tracking impliziter Flüsse[4] verwendet; über das pc-Label kann festgestellt werden, welche Instruktion wann ausgeführt wurde [9]. Die partielle Funktion $\iota \vdash \mu_1\ [\sigma_1]\ pc_1 \xrightarrow{\alpha} \mu_2\ [\sigma_2]\ pc_2$, die eine Maschine von einem Maschinenzustand in den nächsten versetzt, wird Schrittrelation genannt. Die Output-Action α kann entweder τ („silent action“ , kein Output) sein oder ein Atom (Output, der nicht τ ist, wird als Event e bezeichnet). Die Schrittrelation (siehe Appendix B) adaptiert die IFC pur-dynamisch auf einer Low-Level-Maschine. Werden mehrere Label verschiedener Sicherheitsstufen berührt, ist das Ergebnislabel immer auf der höchsten der berührten Sicherheitsstufen.

Bsp.: Die Add-Instruktion (Abbildung 1) addiert zwei mit Labeln versehene Werte. Besitzen die Label verschieden hohe Sicherheitsstufen („low“, „high“), besitzt das Ergebnislabel durch Joining die höhere Sicherheitsstufe der beiden Label („high“).

$$\frac{\iota(n) = Add}{\mu\ [n_1@L_1, n_2@L_2, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [(n_1 + n_2)@L_1 \vee L_2, \sigma]\ (n+1)@L_{pc}}$$

Abbildung 1: abstrakte Add-Schrittregel

4.2 Symbolische Maschine

Die symbolische IFC-Regel-Maschine folgt dem üblichen Ansatz, die IFC-Behandlung auszulagern. Durch die Auslagerung ist es einfacher, Fehler zu erkennen (fehlende Sicherheitsüberprüfung, Taint-Checking[5]), was es wiederum einfacher macht in Coq zu zeigen, dass die IFC-Mechanismen korrekt funktionieren. [10] Zusätzlich wird eine DSL[6] zur Beschreibung der symbolischen IFC-Regeln definiert.

Symbolische Schrittregeln Die symbolische Maschine erhält eine neue IFC-Regeltabelle $\mathcal{R}$. Jede Schrittregel der Regeltabelle enthält einen Aufruf der IFC-Ausführungsrelation (engl.: IFC enforcement relation). Die IFC-Ausführungsrelation prüft, ob eine Instruktionsausführung erlaubt ist und erschafft die Labels für den resultierenden PC (L_{rpc}) und die resultierenden Variablen(L_r) der Instruktion.
Symbolische Schrittregeln haben die Form $\vdash_{\mathcal{R}} (L_{pc}, L_1, L_2, L_3) \rightsquigarrow_{opcode} L_{rpc}, L_r$.
Anhand der Regeltabelle $\mathcal{R}^{abstract}$ (siehe Appendix C), die die symbolischen Schrittregeln mit den IFC-Mechanismen der abstrakten Maschine verknüpft, können die Schrittregeln abgeleitet werden. Eine DSL-Regel besteht aus drei Ausdrücken. Mit *allow* wird angegeben, ob die Instruktion erlaubt ist, e_{rpc} stellt das resultierende pc-Label dar und e_r das Ergebnislabel. Dabei korrespondieren die Labelvariablen LAB_1, LAB_2, LAB_3 und LAB_{pc} zu L_1, L_2, L_3 und L_{pc}.

[4] Implizite Flüsse treten auf, wenn öffentliche Variablen in Abhängigkeit von privaten Variablen verändert werden. [8]
[5] Durch Taint-Checks werden Variablen, die von außen verändert und ein Sicherheitsrisiko darstellen können, gekennzeichnet.
[6] DSL bedeutet domänenspezifische Sprache (engl.: domain-specific language). Eine DSL ist im Gegensatz zu einer general-purpose-Programmiersprache auf ein spezifisches Problem (Domäne) angepasst.

Bsp.: Die DSL-Add-Regel (Abbildung 2) gibt die Semantik der Add-Schrittregel (Abbildung 3) vor. Die Add-Instruktion ist laut $\mathcal{R}^{abstract}$ immer erlaubt. Das *pc*-Label verbleibt unverändert (LAB_{pc}). Das Ergebnislabel besteht aus der Verknüpfung (Join) der beiden Eingabelabels: $LAB_1 \sqcup LAB_2$.

opcode	*allow*	e_{rpc}	e_r
add	*true*	LAB_{pc}	$LAB_1 \sqcup LAB_2$

Abbildung 2: DSL-Add-Regel aus Regeltabelle $\mathcal{R}^{abstract}$

$$\frac{\iota(n) = Add \qquad \vdash_{\mathcal{R}} L_{pc}, L_1, L_2; \_ \rightsquigarrow_{Add} \; L_{rpc}, L_r}{\mu\ [n_1@L_1, n_2@L_2, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [(n_1+n_2)@L_r, \sigma]\ (n+1)@L_{rpc}}$$

Abbildung 3: symbolische Add-Schrittregel

Die Add-Schrittregel besitzt nur drei Inputs (L_{pc}, L_1, L_2). Die Nichtbenutzung von L_3 wird durch _ erkennbar gemacht.

4.3 Konkrete Maschine

Die konkrete Maschine besteht aus der physisch vorhandenen generischen Hardware. Statt einem Labeling wird in der konkreten Maschine ein Tagging der Form $a = n@T$ mit Integerwert-Tags T vorgenommen (siehe Kapitel 2). Die konkrete Maschine benutzt ein Default-Tag T_D mit einem festen Wert (z.B. -1) für Tags, deren Wert irrelevant ist. Die konkrete Aktion (der Output, „concrete action“) α kann τ oder das konkrete Atom a sein.
Für eine Behandlung von Regel-Cache-Misses existiert der Kernel-Modus k; im Benutzermodus u hingegen werden Benutzerprogramme ausgeführt. Die konkrete Maschine verfügt neben dem Benutzerinstruktionsspeicher ι, dem Benutzerdatenspeicher μ, dem PC und dem Stack σ über den Kernel-Instruktionsspeicher ϕ und den Kernel-Datenspeicher κ. Das Privilegienbit π des Maschinenzustands gibt an, in welchem Modus sich die Maschine befindet ($\pi = \{k, u\}$). Im Benutzermodus werden Instruktionen anhand des PCs als Index von ι nachgeschlagen. Die Instruktionen Load und Store benutzen μ. Im Kernel-Modus dient der PC als Index von ϕ, während Load und Store κ benutzen.
Die einzige Regelinstanz im Regel-Cache der konkreten Maschine besteht aus einem Integerwert *opcode*, dem PC-Tag T_{pc}, den Argument-Tags $T_1, ..., T_3$, dem resultierenden PC-Tag T_{rpc} und dem Ergebnis-Tag T_r.

Bsp.: Die Regelinstanz der Add-Instruktion (Tabelle 1) könnte folgendermaßen aussehen:

opcode	T_{pc}	T_1	T_2	T_3	T_{rpc}	T_r
add	0	1	1	-1	0	1

Tabelle 1: Regelinstanz der Add-Instruktion im Regel-Cache

Es seien 0 das Tag für das Label $\bot$, 1 das Tag für das Label $\top$ und -1 das Default-Tag T_D.
Bedeutung: Ist die nächste Instruktion *add*, der PC $\bot$ und die zwei benötigten Argumente $\top$ (nicht benötigte Argumente werden mit T_D gefüllt), dann ist die Instruktion erlaubt. Das Label des neuen PCs ist dann $\bot$ und das Ergebnis-Label $\top$.
Anmerkung: Der Tag-Teil der Atome aus Tabelle 1 ist im Regel-Cache T_D und wird hier nicht angezeigt.

Cache-Hit und Cache-Miss Es existieren zwei Schrittregelsätze im Benutzermodus. Im Fall des Cache-Hit (Abbildung 4) wird die Instruktion normal ausgeführt. Sollte ein Cache-Miss (Abbildung 5)

auftreten, werden der *opcode* sowie die Tags vom PC und der Argumente in der Cache-Zeile gespeichert. Die Maschine simuliert dann einen Aufruf des Cache-Fault-Handlers.

Bsp. für die Add-Instruktion im Fall des Cache-Hit:

$$\frac{\iota(n) = Add \qquad \kappa = \boxed{add \mid T_{pc} \mid T_1 \mid T_2 \mid T_D \parallel T_{rpc} \mid T_r}}{u\ \kappa\ \mu\ [n_1@T_1, n_2@T_2, \sigma]\ n@T_{pc} \xrightarrow{\tau} u\ \kappa\ \mu\ [(n_1+n_2)@T_r, \sigma]\ (n+1)@T_{rpc}}$$

Abbildung 4: Cache-Hit

T_1 und T_2 sind die beiden Tags der zwei obersten Atome auf dem Stack. Nach Instruktionsausführung wird T_r auf den Stack gepusht.

Bsp. für die Add-Instruktion im Fall des Cache-Miss:

$$\frac{\iota(n) = Add \qquad \kappa_i \neq \boxed{add \mid T_{pc} \mid T_1 \mid T_2 \mid T_D} = \kappa_j}{u\ [\kappa_i, \kappa_o]\ \mu\ [n_1@T_1, n_2@T_2, \sigma]\ n@T_{pc} \xrightarrow{\tau} k\ [\kappa_j, \kappa_D]\ \mu\ [(n@T_{pc}, u); n_1@T_1, n_2@T_2, \sigma]\ 0@T_D}$$

Abbildung 5: Cache-Miss

Sei κ_i der Input-Teil und κ_o der Output-Teil des Regel-Caches. κ_i stimmt beim Cache-Miss nicht mit dem übergebenen Input überein und der Cache-Fault-Handler wird aufgerufen. Der nächste Maschinenzustand wird gebildet, indem zuerst der Input-Teil des Caches in die verlangte Form κ_j und der Output-Teil in die Form $\kappa_D \stackrel{\triangle}{=} \boxed{T_D \mid T_D}$ gebracht wird. Danach wird ein neuer Return-Rahmen auf den Stack gepusht, um das Privileg-Bit (u) und den PC zu speichern. Das Privileg-Bit bekommt nun den Wert k, damit die nächste Instruktion aus dem Kernel-Instruktionsspeicher gelesen wird und der PC auf 0 gesetzt. An Position 0 beginnt die Cache-Fault-Handler-Routine im Kernel-Instruktionsspeicher.
Der Cache-Fault-Handler soll nun entweder das entsprechend resultierende PC-Tag und das Ergebnis-Tag in den Kernel-Instruktionsspeicher schreiben und einen Rücksprung (Ret-Instrukion) ausführen, damit die Instruktion im Benutzermodus neu gestartet wird oder, falls die Instruktion durch eine ungültige Kombination von *opcode* und Argument-Tags nicht ausgeführt werden darf, die Maschine anhalten, indem zu einem ungültigen PC (-1) gesprungen wird.
Im Kernel-Modus wird der Regel-Cache nicht eingesehen, um infiniten Regress[7] zu vermeiden. Die meisten Tags werden ignoriert; Tags des neuen Zustands erhalten den Standard-Tag T_D.

Bsp.: Add- und Ret-Schrittregel im Kernel-Modus der konkreten Maschine:

$$\frac{\phi(n) = Add}{k\ \kappa\ \mu\ [n_1@\_, n_2@\_, \sigma]\ n@\_ \xrightarrow{\tau} k\ \kappa\ \mu\ [(n_1+n_2)@T_D, \sigma]\ n+1@T_D}$$

Für einen Rücksprung vom Kernel-Modus in den Benutzermodus werden aus dem Return-Rahmen Privileg-Bit und PC mitsamt dessen Tag vom Stack genommen:

$$\frac{\phi(n) = Ret}{k\ \kappa\ \mu\ [(n'@T_1, \pi); \sigma]\ n@\_ \xrightarrow{\tau} \pi\ \kappa\ \mu\ [\sigma]\ n'@T_1}$$

5 Cache-Fault-Handler

Damit die konkrete die symbolische Maschine implementieren kann, muss ein Cache-Fault-Handler im Kernel-Instruktionsspeicher der konkreten Maschine installiert sein. Der Handler simuliert Look-

[7] Ein infiniter Regress bedeutet, dass eine Kaskade von unendlich sich bedingenden Bedingungen vorliegt.

Up und Evaluierung der DSL-Regeln der symbolischen Maschine, welche direkt zu Maschinencode kompiliert werden.

5.1 Funktionsaufrufe des Cache-Fault-Handlers

Die *gen*∗-Funktionen generieren eine Liste von konkreten Maschineninstruktionen, die durch die Regeltabelle $\mathcal{R}$ parametrisiert sind (siehe Appendix C für $\mathcal{R}^{abstract}$). Die Cache-Fault-Handler-Routine wird durch den Generator *genFaultHandler* eingeleitet (Tabelle 2).

genFaultHandler $\mathcal{R}$ =	genComputeResults $\mathcal{R}$ ++ genStoreResults ++ genIf [Ret] [Push(-1); Jump]
genComputeResults $\mathcal{R}$ =	genIndexedCases[] genMatchOp(genApplyRule ∘ $Rule_{\mathcal{R}}$) opcodes
genMatchOp *op* =	[Push *op*] ++ genLoadFrom addrOpLabel ++ genEqual
genApplyRule $\langle allow e_{rpc}, e_r \rangle$ =	genBool *allow* ++ genIf(genSome(genELab e_{rpc} ++ genELab e_r)) genNone
genELab *BOT* =	genBot
genELab Lab_i =	genLoadFrom addrTag$_i$
genELab $LE_1 \sqcup LE_2$ =	genELab LE_1 ++ genELab LE_2 ++ genJoin
genBool *TRUE* =	genTrue
genBool $LE_1 \sqsubseteq LE_2$ =	genELab LE_1 ++ genELab LE_2 ++ genFlows
genStoreResults =	genIf(genStoreAt addrTag$_r$ ++ genStoreAt addrTag$_{rpc}$ ++ genTrue) genFalse
genIndexedCases *genDefault genGuard genBody* =	*g*
mit *g* nil =	*genDefault*
$g(n :: ns)$ =	*genGuard n* ++ genIf (*genBody n*) (g n s)
genIf t f =	genSkipIf (length f') ++ f' ++ t mit $f' = f$ ++ genSkip(length t)
genSkip n =	genTrue ++ genSkipIf n
genSkipIf n =	[Bnz$(n+1)$]

Tabelle 2: Fault-Handler-Generierung der IFC-Regeltabelle

Die *addr*∗-Parameter geben die Positionen des Opcodes und des Tags im Regel-Cache der konkreten Maschine an. Der Wert $false$ wird durch 0 repräsentiert, *true* durch einen Wert, der von 0 verschieden ist. Der Cache-Fault-Handler durchläuft bei seiner Ausführung drei Phasen:

Phase 1 *genComputeResults* besteht aus einer verschachtelten if-then-else-Anweisung: *genIndexedCases* vergleicht den Opcode der Instruktion, die den Cache-Miss ausgelöst hat gegen sämtliche andere Opcodes via *genMatchOp* (der gepushte Opcode wird gegen einen neu geladenen Opcode verglichen, indem beide voneinander subtrahiert werden [11]). Wurde ein passender Opcode gefunden, wird der generierte Code ausgeführt, der zur symbolischen IFC-Regel passt. Die Ergebnisse des generierten Codes werden mit *genApplyRule* auf den Stack gepusht. Sollte die Instruktion erlaubt sein, werden PC- und Ergebnis-Tags auf den Stack gepusht.

Phase 2 *genStoreResults* liest die Ergebnisse im Stack aus und passt den Regel-Cache an. Induziert das Ergebnis, dass die Instruktion erlaubt ist, werden der resultierende PC und die Ergebnis-Tags in den Cache geschrieben und *true* auf den Stack gepusht. Ist die Instruktion nicht erlaubt, wird nur $false$ auf den Stack gepusht.

Phase 3 Der boolesche Wert auf dem Stack wird ausgelesen: Im Fall *true* wird zum Benutzermodus zurückgekehrt, andernfalls zur Adresse -1 gesprungen und die Maschine angehalten.

Zweipunktverband (boolescher Verband) Verbandspezifische[8] Generatoren parametrisieren implizit die Label-Ausdrücke *genELab* und die booleschen Ausdrücke *genBool*. Um die Generatoren *genBot*, *genJoin* und *genFlows* zu implementieren, muss eine Darstellung der abstrakten Labels als Integer-Tags und eine Instruktionssequenz für die Kodierung gewählt werden (konkreter Verband, engl.: concrete lattice). Das Tupel $CL = (Tag, Lab, genBot, genJoin, genFlows)$ beschreibt den konkreten Verband formal.
Im booleschen Verband können Labels als boolesche Werte kodiert werden ($\bot$ als 0 und $\top$ als Wert verschieden von 0). Dann gibt genBot $false$ zurück, genJoin verknüpft die Label (führt einen Join aus) und genFlows vergleicht Label entsprechend der Ordnungsrelation des Verbands [7, 12].

genIf und genIndexedCases *genIf* bekommt zwei Code-Sequenzen als Parameter übergeben. Die erste Sequenz wird ausgeführt, wenn eine Bedingung wahr ist; die zweite, sofern die Bedingung falsch ist. Der erste Wert auf dem Stack wird überprüft und entsprechend die Kontrolle weitergegeben.
genIndexedCases erhält als Parameter eine Liste von Integerwerten und Funktionen. Daraus werden Funktionsrümpfe und Guards generiert. Die Guards werden der Reihe nach ausgeführt, bis *true* von einem der Guards zurückgegeben wird und der entsprechende Funktionsrumpf ausgeführt.

6 Korrektheit des Cache-Fault-Handlers

Seien $\mathcal{R}$ eine symbolische Regeltabelle, $\phi_{\mathcal{R}} \stackrel{\triangle}{=} genFaultHandler\ \mathcal{R}$ der dazugehörige Cache-Fault-Handler und $\kappa_i = \boxed{opcode \mid \text{Tag}(L_{pc}) \mid \text{Tag}(l_1) \mid \text{Tag}(l_2) \mid \text{Tag}(l_3)}$. Die transitive Hülle $\phi \vdash cs_1 \rightarrow_k^* cs_2$ einer konkreten Schrittrelation definiert die Relation zwischen dem Zustand eines Einstiegspunkts und eines Endzustands der Cache-Fault-Handler-Routine. Alle Vorgängerzustände von cs_2 haben $\pi = k$ gesetzt.
Diese Korrektsheitsaussage wird von den folgenden zwei Lemmata erfasst.

Lemma 6.1 Fault-Handler-Korrektheit im erlaubten Fall:
Angenommen $\vdash_{\mathcal{R}} (L_{pc}, l_1, l_2, l_3) \rightsquigarrow_{opcode} L_{rpc}, L_r$, dann
$\phi_{\mathcal{R}} \rightsquigarrow \langle k\ [\kappa_i, \kappa_o]\ \mu\ [(pc, u); \sigma]\ 0@T_D\rangle \rightarrow_k^* \langle u\ [\kappa_i, \kappa_o']\mu\ [\sigma]\ pc\rangle$
mit Cache-Output $\kappa_o' = Tag(L_{rpc}, Tag(L_r))$.

Lemma 6.2 Fault-Handler-Korrektheit im nicht erlaubten Fall:
Angenommen $\vdash_{\mathcal{R}} (L_{pc}, l_1, l_2, l_3) \not\rightsquigarrow_{opcode}$, dann
$\phi_{\mathcal{R}} \rightsquigarrow \langle k\ [\kappa_i, \kappa_o]\ \mu\ [(pc, u); \sigma]\ 0@T_D\rangle \rightarrow_k^* \langle k\ [\kappa_i, \kappa_o]\mu\ [\sigma']\ -1@T_D\rangle$
für einen Stack σ'.

Dadurch, dass der Fault-Handler-Code aus Generatoren besteht, genügt es den Korrektheitsbeweis auf die Generatoren zu reduzieren. Es wird ein Standard-Hoare-Tripel[9] $\{P\}\ c\ \{Q\}$ für Code, der keine Sprung- oder Return-Sequenzen enthält (in sich geschlossener Code), definiert. Ebenso wird ein für den letzten Schritt des Fault-Handlers angepasstes Hoare-Tripel $\{P\}\ c\ \{Q\}_{pc}^{O}$ für Escape-Sequenzen bestimmt. Seien P dabei der Anfangszustand, c der ausgeführte Code und Q der Endzustand.

6.1 In sich geschlossene Code-Sequenzen

Seien P und Q Prädikate auf $\kappa \times \sigma$ und c eine Code-Sequenz. Ist c die Code-Sequenz des derzeitigen pcs und erfüllen κ und σ P, dann ist Q der Zustand nach Ausführung von c. Der pc zeigt dann auf eine nächste Code-Sequenz und die resultierenden κ' und σ' erfüllen Q:

$$\{P\}\ c\ \{Q\} \stackrel{\triangle}{=} \qquad c = \phi(n), ..., \phi(n'-1) \wedge P(\kappa, \sigma) \Longrightarrow$$
$$\exists \kappa', \sigma'.Q(\kappa', \sigma') \wedge \phi \vdash \langle k\ \kappa\ \mu\ [\sigma]n@T_D\rangle \rightarrow_k^* \langle k\ \kappa'\ \mu\ [\sigma']n'@T_D\rangle$$

[8] Ein Verband ist eine Ordnungsstruktur, in der je zwei Elemente ein Supremum und ein Infimum haben.
[9] Das Hoare-Kalkül erlaubt es, die Korrektheit imperativer Programme anhand logischer Regeln zu bestätigen.

Durch die Fähigkeit des Hoare-Kalküls, partielle Korrektheit nachzuweisen und durch den Determinismus der konkreten Maschine, kann die totale Korrektheit nachgewiesen, d.h. verifiziert werden. Die Tripel werden im Stil des wp-Kalküls („weakest-precondition-Kalkül“: Alternative zum Hoare-Kalkül.) angegeben.

Code-Verifikation
Beweise werden für jede Instruktion und jeden Generator bzw. für jede Verbandoperation erstellt.

Bsp.: Die Add-Instruktion wird folgendermaßen spezifiziert:

$$\frac{P(\kappa,\sigma) := \exists n_1\ T_1\ n_2\ T_2\ \sigma'.\ \sigma = n_1@T_1, n_2@T_2, \sigma' \wedge Q(\kappa, ((n_1+n_2)@T_D, \sigma'))}{\{P\}\ [\text{Add}]\ \{Q\}}$$

6.2 Code mit Sprung- oder Return-Anweisung

Das Tripel $\{P\}\ c\ \{Q\}^O_{pc}$ mit der Funktion O, die von $\kappa \times \sigma$ auf ein Ergebnis (Outcome) *Erfolg* oder *Misserfolg* abbildet, spezifiziert Code, der entweder genau einen Sprung aus c macht, oder in den Benutzermodus zurückkehrt. Enthält ϕ die Code-Sequenz c am derzeitigen pc und erfüllen κ und σ P, dann ist Q erfüllt,
i. wenn O *Erfolg* berechnet. (Die Maschine arbeitet solange im Kernel-Modus, bis sie zum Benutzermodus zurückspringt.)
ii. wenn O *Misserfolg* berechnet. (Die Maschine stoppt im Kernel-Modus bei $PC = -1$.)
Um die Jump- und die Ret-Instruktion (siehe Abbildung 6) zu spezifizieren und somit die in sich geschlossenen Code-Sequenzen mit Jump- und Ret-Instruktionen sowie dem letzten *genIf* zu koppeln und die Verifizierung des Cache-Fault-Handlers abzuschließen, werden zwei weitere Tripel definiert:

$$\frac{\{P_1\}\ c_1\ \{P_2\} \qquad \{P_2\}\ c_2\ \{P_3\}^O_{pc}}{\{P_1\}\ c_1{+}{+}c_2\ \{P_3\}^O_{pc}} \qquad \frac{\{P\}\ c_1\ \{Q\}^O_{pc}}{\{P\}\ c_1{+}{+}c_2\ \{Q\}^O_{pc}}$$

Bsp. für die Ret-Instruktion:

$$\frac{\begin{array}{l} P(\kappa,\sigma) := \exists \sigma'.\ Q(\kappa,\sigma') \wedge \sigma = (pc,u);\sigma' \\ O(\kappa,\sigma) := \mathit{Erfolg} \end{array}}{\{P\}\ [\text{Ret}]\ \{Q\}^O_{pc}}$$

Abbildung 6: Spezifikation der Ret-Instruktion

7 Verfeinerung zwischen abstrakter und konkreter Maschine

Um zu zeigen, dass die konkrete Maschine die Nichtinterferenz-Eigenschaft besitzt (siehe Kapitel 8), muss vorerst gezeigt werden, dass die konkrete die symbolische Maschine verfeinert. Es wird außerdem gezeigt, dass dies auch umgekehrt gilt; somit implementiert die konkrete die abstrakte Maschine.
Für die folgenden Beweisskizzen wird eine Maschine zuerst formal definiert:

Definition 7.1 Generische Maschine:
Sei $M = (S, E, I, \cdot \dot{\longrightarrow} \cdot, \mathit{Init})$ eine Maschine mit Zuständen $s \in S$, Events $e \in E$, Input-Daten $i \in I$, einer Schrittrelation $\cdot \dot{\longrightarrow} \cdot \subseteq S \times (E + \{\tau\}) \times S$ mit Output $\alpha \in E + \{\tau\}$. Sei $\mathit{Init} \in E \to S$ eine Funktion, die Initialzustände der Maschine via der Inputdaten I erschaffen kann.

Init erzeugt beim Laden des Programms p einen initialen Stack und Speicher, sowie den PC. Ein generischer Schritt $s_1 \xrightarrow{\alpha} s_2$ erzeugt ein Event e oder $\alpha = \tau$. Die reflexiv-transitive Hülle, die Traces produziert, vernachlässigt stille Schritte ($s_1 \xrightarrow{t}{}^* s_2$). Irrelevante Endzustände werden ebenfalls nicht mit aufgeschrieben: ($s \xrightarrow{e}$ bzw. $s \xrightarrow{t}{}^*$).

7.1 Relation zwischen verschiedenen Maschinen

Um zwei verschiedene Maschinen in einen Zusammenhang zu bringen, werden deren Traces analysiert. Die Relation zwischen Events $\rhd \subseteq E_1 \times E_2$ kann auf Aktionen α und Traces übertragen werden:
$\alpha_1[\rhd]\alpha_2 \quad \triangleq \quad (\alpha_1 = \tau = \alpha_2 \vee \alpha_1 = e_1 \rhd e_2 = \alpha_2)$
$\vec{x}\ [\rhd]\ \vec{y} \quad \triangleq \quad length(\vec{x}) = length(\vec{y}) \wedge \forall i.x_i \rhd y_i$
Bei einer Relation zwischen Aktionen sind entweder beide Aktionen still (τ), d.h. es existiert kein beobachtbarer Output oder die Aktionen gleichen sich durch die Relation zwischen ihren Events. Bei einer Relation zwischen Traces sind nach Definition die jeweiligen Längen der Traces gleich.

Definition 7.2 Verfeinerung:
Seien $M_1 = (S_1, E_1, I_1, \cdot \dot{\longrightarrow} \cdot, Init_1)$ und $M_2 = (S_2, E_2, I_2, \cdot \dot{\longrightarrow} \cdot, Init_2)$ zwei Maschinen. Die Verfeinerung von M_1 zu M_2 (M_2 verfeinert M_1) ist ein Relationspaar $(\rhd_i, \rhd_e)$ mit $\rhd_i \subseteq I_1 \times I_2$, $\rhd_e \subseteq E_1 \times E_2$, sodass wenn $i_1 \ \rhd_i \ i_2$ und $Init_2(i_2) \xrightarrow{t_2}{}^*$ gelten, ein Trace t_1 existiert, sodass $Init_1(i_1) \xrightarrow{t_1}{}^*$ und $t_1[\rhd_e]t_2$ gelten.

Für den Beweis der Maschinenverfeinerung wird noch die Verfeinerung durch Zustände definiert.

Definition 7.3 Zustandsverfeinerung:
Seien M_1, M_2 wie in Definition 7.2 gegeben. Eine Zustandsverfeinerung von M_1 zu M_2 (M_2 verfeinert M_1) ist ein Relationspaar $(\rhd_s, \rhd_e)$ mit $\rhd_s \subseteq S_1 \times S_2$, $\rhd_e \subseteq E_1 \times E_2$, sodass wenn $s_1 \rhd_s s_2$ und $s_2 \xrightarrow{t_2}{}^*$ gelten, ein Trace t_1 existiert, sodass $s_1 \xrightarrow{t_1}{}^*$ und $t_1[\rhd_e]t_2$ gelten.

Ist die Inputrelation mit der Zustandsrelation kompatibel, kann eine Verfeinerung mit der Zustandsverfeinerung bewiesen werden.

Lemma 7.4 Angenommen $i_1 \rhd_i i_2 \Rightarrow Init_1(i_1) \rhd_s Init_2(i_2)$ für alle i_1, i_2. Wenn $(\rhd_s, \rhd_e)$ eine Zustandsverfeinerung ist, dann ist $(\rhd_i, \rhd_e)$ eine Verfeinerung.

7.2 Verfeinerung von abstrakter durch symbolische Maschine

Dadurch, dass die symbolische Maschine die gleichen Schrittrelationen und die gleichen Eingabedaten wie die abstrakte Maschine besitzt, sind auch deren Traces gleich.

Definition 7.5 Abstrakte und symbolische Maschine als generische Maschinen:
Die Eingabedaten beider Maschinen bestehen aus einem 4-Tupel $(p, args, n, L)$ mit p Programm, $args$ Liste von Atomen (initialer Stack) und n (Speichergröße) Kopien von $0@L$ (initialer PC).

Lemma 7.6 Ist die symbolische Maschine mit Regeltabelle $\mathcal{R}^{abstract}$ instantiiert, verfeinert die symbolische die abstrakte Maschine durch $(=, =)$.

7.3 Verfeinerung von symbolischer durch konkrete Maschine

Seien eine feste aber willkürliche Regeltabelle $\mathcal{R}$, ein konkreter Tag-Verband und ein abstrakter Label-Verband, sowie die Korrektheit des Cache-Fault-Handlers gegeben. Darüber hinaus kann angenommen werden, dass der Cache-Fault-Handler der konkreten Maschine zu den Regeln der symbolischen korrespondiert, d.h., dass $\phi = \phi_{\mathcal{R}}$ und die Integer-Tag-Kodierung der Label korrekt ist.

Definition 7.7 konkrete Maschine als generische Maschine:
Die Eingabedaten der konkreten Maschine bestehen aus einem 4-Tupel $(p, args, n, T)$ mit p Programm, $args$ Liste mit Atomen (initialer Stack) und n (Speichergröße) Kopien von $0@T$ (initialer PC).

Die beiden Maschinen besitzen unterschiedliche Eingabedaten und Events. Durch die Relationen $\rhd_i^c$ und $\rhd_e^c$ werden sie aneinander angepasst. Dabei sind Payloadwerte gleich und die Labels entsprechen den Tags modulo des Funktionstags des konkreten Verbands:

$$\frac{args' = map(\lambda(n@L).n@Tag(L))args}{(p, args, n, L) \rhd_i^c (p, args', n, Tag(L))} \qquad \frac{}{n@L \rhd_e^c n@Tag(L)}$$

Theorem 7.8 Die konkrete verfeinert die symbolische Maschine durch $(\rhd_i^c, \rhd_e^c)$.

Der Beweis von Theorem 7.8 erfolgt durch Zustandsverfeinerung (Lemma 7.11). Sei im Folgenden $\rhd_s^c$ die Passungs- bzw. Match-Relation der Zustände von konkreter und symbolischer Maschine. Es werden nur konkrete Benutzermodus-Zustände gematcht [2].

Lemma 7.9 Verfeinerung - erfolgreicher konkreter Schritt:
Sei cs_1^u ein konkreter Zustand und man nehme an, dass $cs_1^u \xrightarrow{\alpha_c} cs_2^u$ mit $u \in \pi$. Sei qs_1 ein symbolischer Zustand und $qs_1 \rhd_s^c cs_1^u$. Dann $\exists\ qs_2, \alpha_a$, sodass $qs_1 \xrightarrow{\alpha_a} qs_2$ mit $qs_2 \rhd_s^c cs_2^u$ und $\alpha_a[\rhd_e^c]\alpha_c$.

Lemma 7.10 Verfeinerung - fehlgeschlagener konkreter Schritt:
Sei cs_0^u ein konkreter Zustand und cs_1^k der Zustand, nachdem ein Cache-Miss aufgetreten ist und die Maschine in den Kernel-Modus wechselt. Sei cs_n^k der letzte Zustand im Kernel-Modus, bevor die Maschine wieder in den Benutzermodus (Zustand cs_{n+1}^u) zurückkehrt. Sei qs_0 ein zu cs_0^u korrespondierender symbolischer Zustand. Dann gilt $qs_0 \rhd_s^c cs_{n+1}^u$.

Für einen Beweis von Lemma 7.10, werden Lemma 6.1 (erlaubte Operation) und Lemma 6.2 (nicht erlaubte Operation) wiederholt angewandt, um ein Match zwischen konkretem Zustand und Ausführung symbolischer Regeln zu erhalten.
Endet die Ausführung ohne eine Rückkehr in den Benutzermodus (Sprung zu ungültigem PC), werden keine Ausgabewerte produziert, sodass auch in diesem Fall die Traces der konkreten und der symbolischen Maschine übereinstimmen.

Lemma 7.11 $(\rhd_s^c, \rhd_e^c)$ definiert eine Verfeinerung zwischen symbolischer und konkreter Maschine durch Zustände.

7.4 Verfeinerung zwischen abstrakter und konkreter Maschine

Aus Lemma 7.6 und Theorem 7.8 lässt sich schlussfolgern, dass die konkrete die abstrakte Maschine verfeinert.
Für den umgekehrten Fall werden die inversen Relationen von $\rhd_i^c$ und $\rhd_e^c$ ($\rhd_i^{-c}$ respektive $\rhd_e^{-c}$) benutzt:

Theorem 7.12 Inverse Verfeinerung:
Die abstrakte verfeinert die konkrete Maschine via $(\rhd_i^{-c}, \rhd_e^{-c})$.

Durch den Beweis von Theorem 7.12 wird sichergestellt, dass die konkrete Maschine tatsächlich das Verhalten der abstrakten Maschine reflektiert und die Traces der abstrakten auch von der konkreten Maschine ausgegeben werden. Die symbolische Maschine dient wieder als Zwischenschritt.

Lemma 7.13 Vorwärts-Verfeinerung:
Seien qs_0 und cs_0 zwei Zustände mit $cs_0 \rhd_s^{-c} qs_0$. Angenommen die symbolische Maschine führt einen Schritt $qs_0 \xrightarrow{\alpha_c}{}^* qs_1$ aus. Dann existieren ein Zustand cs_1 und Output α_c, sodass $cs_0 \xrightarrow{\alpha_c}{}^* cs_1$ gilt mit $cs_1 \rhd_s^{-c} qs_1$ und $\alpha_c[\rhd_e^{-c}]\alpha_a$.

Für einen Beweis werden die beiden Fälle des Cache-Hit/Miss analysiert: Tritt ein Cache-Hit auf (Cache-Input matcht opcode und data von cs_0), kann der konkrete Schritt $cs_0 \xrightarrow{\alpha_c} cs_1$ ausgeführt

werden.
Durch $qs_0 \xrightarrow{\alpha_a} qs_1$ und die Anwendung von Lemma 6.1 kann der konkrete Schritt bei einem Cache-Miss ausgeführt werden.

8 Nichtinterferenz

Für den Beweis der Nichtinterferenz muss zunächst der Trace in Bezug auf Beobachtbarkeit definiert werden.

Definition 8.1 Beobachtbarkeit (engl.: Observation):
Die Beobachtbarkeit eines Output-Trace wird als 3-Tupel $(\Omega, \lfloor\cdot\rfloor_\cdot, \cdot \approx_\cdot \cdot)$ definiert. Sie besteht aus einer Menge von Beobachtern (engl.: Observer) Ω. Für alle $o \in \Omega$ ist $\lfloor\cdot\rfloor_o \subseteq E$ ein Prädikat der Beobachtbarkeit von Beobachter-Events und $\cdot \approx_o \cdot \subseteq I \times I$ eine Ununterscheidbarkeitsrelation von Input-Daten des Beobachters o.

$\lfloor t \rfloor_o$ ist ein Trace, in welchem alle unbeobachtbaren Events aus t mit $\lfloor\cdot\rfloor_o$ gefiltert werden. $t_1 \approx t_2$ besagt, dass die Traces t_1 und t_2 ununterscheidbar sind. Die verschieden langen Traces werden so gekürzt, dass sie eine gleiche Länge besitzen:
Angenommen t_1 ist der längere und t_2 der kürzere Trace. Dann wird t_1 an der Stelle $length(t_2)$ abgeschnitten. Die Elemente von t_2 sind mit denen von t_1 nach Verkürzung paarweise identisch.

TINI Die **T**ermination-**I**nsensitive **N**on**I**nterference-Eigenschaft ist die Eigenschaft der Nichtinterferenz, unter der Akzeptanz, dass ein Leck im Informationsfluss abhängig vom Terminierungsverhalten eines Programms entstehen kann. [13] TINI wird im Folgenden formal definiert:

Definition 8.2 TINI:
Sei eine generische Maschine nach Definition 7.1 gegeben. Eine solche beobachtete Maschine besitzt die TINI-Eigenschaft, wenn für alle $o \in \Omega$, für alle ununterscheidbaren Paare $i_1 \approx_o i_2$ und für alle Paare $Init(i_1) \xrightarrow{t_1} *$ und $Init(i_2) \xrightarrow{t_2} *$ gilt, dass $\lfloor t_1 \rfloor_o \approx \lfloor t_2 \rfloor_o$.

Die Init-Funktionen $Init(i_1) \xrightarrow{t_1} *$ und $Init(i_2) \xrightarrow{t_2} *$ können mit den Input-Werten i_1 und i_2 evtl. unterschiedlich lange Traces erzeugen. Ein Beobachter kann aufgrund der Trace-Verkürzung nicht zwischen unproduktiver Schleife, System-Stopp oder erfolgreicher Terminierung unterscheiden.

8.1 TINI der abstrakten Maschine

Definition 8.3 Beobachtbarkeit für die abstrakte Maschine:
Sei $\mathcal{L}$ ein Verband mit Ordnungsrelation $\leq$. Die Ununterscheidbarkeit von Atomen $a_1 \approx_o^a a_2$ ist definiert als:

$$\text{i} \quad \frac{}{a \approx_o^a a} \qquad\qquad \text{ii} \quad \frac{\neg\lfloor a_1 \rfloor_o \quad \neg\lfloor a_2 \rfloor_o}{a_1 \approx_o^a a_2}$$

Die Beobachtbarkeit ist als $(\mathcal{L}, \lfloor\cdot\rfloor_\cdot^a, \cdot \approx_\cdot^a \cdot)$ mit $\lfloor n@L \rfloor_o^a \triangleq L \leq o$ definiert.
Für Input-Daten der abstrakten Maschine gilt $(p, args_1, n, L) \approx_o^a (p, args_2, n, L) \triangleq args_1[\approx_o^a] args_2$.
$[\approx_o^a]$ ist die Notation der Ununterscheidbarkeit für Listen.

Zu i: Gleiche Atome sind immer ununterscheidbar. Zu ii: Die Atome a_1, a_2 sind ununterscheidbar, wenn o keins der beiden Atome beobachten kann.
Zwei Listen $args_1, args_2$ sind ununterscheidbar, sofern die Input-Daten der abstrakten Maschine ununterscheidbar sind.

Theorem 8.4 Die abstrakte Maschine besitzt die TINI-Eigenschaft.

Die TINI-Eigenschaft für die abstrakte Maschine wird durch Unwinding-Conditions[10] bewiesen.

8.2 Erhaltung von TINI bei Verfeinerung

Theorem 8.5 Wenn Maschine M_2 die Maschine M_1 verfeinert und für jeden Beobachter o_2 für M_2 ein Beobachter o_1 für M_1 existiert und folgende Kompatibilitätsaussagen für

$e_1, e_1' \in E_1$	$e_2, e_2' \in E_2$	$i_2, i_2' \in I_2$

gelten:

i $e_1 \rhd_e e_2 \Rightarrow (\lfloor e_1 \rfloor_{o_1} \Leftrightarrow \lfloor e_2 \rfloor_{o_2})$

ii $i_2 \approx_{o_2} i_2' \Rightarrow \exists i_1 \approx_{o_1} i_1'.\ (i_1 \rhd_i i_2 \wedge i_1' \rhd_i i_2')$

iii $(e_1 \approx_{o_1} e_1' \wedge e_1 \rhd_e e_2 \wedge e_1' \rhd_e e_2') \Rightarrow e_2 \approx_{o_2} e_2'$

Dann besitzt M_2 die TINI-Eigenschaft, wenn M_1 sie besitzt.

Zu i: Findet eine Verfeinerung von e_1 zu e_2 statt (e_2 verfeinert e_1), dann ist e_1 genau dann von o_1 beobachtbar, wenn e_2 von o_2 beobachtbar ist (siehe auch Kapitel 7.1). Zu ii: Sind die Input-Daten i_2 und i_2' für o_2 ununterscheidbar, dann existieren für o_1 ununterscheidbare i_1, i_1', sodass die i_2 i_1 und i_2' i_1' verfeinert. Zu iii: Sind e_1, e_1' ununterscheidbar und verfeinert e_2 e_1 und e_2' e_1', dann sind auch e_2, e_2' ununterscheidbar.
Sind diese drei Eigenschaften samt der Kompatibilitätsaussagen gegeben, dann bleibt bei einer Verfeinerung von Maschine M_1 durch Maschine M_2 die TINI-Eigenschaft erhalten.

8.3 TINI der konkreten Maschine mit Fault-Handler

Voraussetzung für einen Beweis ist, dass die Verbandoperationen *genBot*, *genJoin* und *genFlows* des abstrakten Verbands $\mathcal{L}$ spezifikationsgetreu in den konkreten Verband CL kodiert wurden.

Definition 8.6 Beobachtbarkeit der konkreten Maschine:
Sei $\mathcal{L}$ ein Verband und CL der dazu korrespondierende und korrekt implementierte konkrete Verband. Die Beobachtbarkeit ist definiert als $(\mathcal{L}, \lfloor\cdot\rfloor^c_\cdot, \cdot \approx^c_\cdot \cdot)$ mit

i $\lfloor n@T \rfloor^c_o \triangleq LAB(T) \leq o$

ii $(p, args_1', n, T) \approx^c_o (p, args_2', n, T) \triangleq args_1[\approx^a_o]args_2$

iii $args_i' = map(fun\ n@L \rightarrow n@Tag(L))\ args_i$

Durch Beweisen der Verfeinerung in Kapitel 7 und den Beweis, dass diese die Kompatibilitätsbeschränkung aus 8.2 erfüllt, kann letzendlich geschlussfolgert werden, dass die konkrete Maschine die TINI-Eigenschaft besitzt.

Theorem 8.7 Die konkrete Maschine mit Cache-Fault-Handler $\phi_{\mathcal{R}^{abstract}}$ besitzt die TINI-Eigenschaft.

9 Resümee und Ausblick

In dieser Seminararbeit wurden die Hard- und Softwarefeatures von SAFE mit einem Fokus auf Verifizierung der Funktionalität analysiert. Dazu wurde die Aufteilung in abstrakte und konkrete Maschine, sowie die Auslagerung der IFC-Mechanismen in die symbolische Maschine eingeführt. Es wurde die Beweismethodik der Verifizierung des Cache-Fault-Handlers der konkreten Maschine dargestellt. Um eine Beweisbasis für die Nichtinterferenz (TINI) von SAFE zu erhalten, wurde die Beweisstrategie der gegenseitigen Verfeinerung der Maschinen behandelt.

2009 konnte verifiziert werden, dass der seL4-Mikrokernel funktional korrekt ist [15]. Durch diesen Beweis inspiriert, liegt die Zukunft des von der DARPA finanzierten [3] (CRASH/)SAFE-Projekts darin, dass das SAFE-Modell auf Mehrkernebene mit Nebenläufigkeit bewiesen werden kann.

[10] Unwinding ist eine Verifikationsmethode für auf Nichtinterferenz basierende Aussagen. Nichtinterferenzeigenschaften können auf einfacher zu beweisende Bedingungen reduziert werden. [14]

In weiteren Untersuchungen soll gezeigt werden, dass Speichersicherheit und Kontrollflussintegrität von SAFE effizient unterstützt werden können.
Es bestehen außerdem Pläne, einige SAFE-Features wie den Cache-Fault-Handler auf traditionellere Architekturen zu portieren. SAFE könnte sich auf eingebetteten System etablieren [4]; evtl. ist eine Zusammenarbeit mit der ebenfalls von der DARPA finanzierten Hochsicherheitsarchitektur MILS möglich. [16]

A Basisinstruktionssatz

instr ::=		Instruktion
	Add	Addition zweier Werte
|	Output	Ausgabe des ersten Elements des Stacks
|	Push n	Push von n auf Stack
|	Load	indirektes Laden aus Datenspeicher
|	Store	indirektes Speichern in Datenspeicher
|	Jump	unbedingter Sprung
|	Bnz n	bedingter Sprung (Branch Not Zero)
|	Call	Aufruf eines Unterprogramms
|	Ret	Rückkehr aus Unterprogramm

B Semantik der abstrakten IFC-Maschine

$$\frac{\iota(n) = Add}{\mu\ [n_1@L_1, n_2@L_2, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [(n_1 + n_2)@L_1 \vee L_2, \sigma]\ (n+1)@L_{pc}}$$

$$\frac{\iota(n) = Output}{\mu\ [m@L_1, \sigma]\ n@L_{pc} \xrightarrow{m@(L_1 \vee L_{pc})} \mu\ [\sigma]\ (n+1)@L_{pc}}$$

$$\frac{\iota(n) = Push\ m}{\mu\ [\sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [m@\bot, \sigma]\ (n+1)@L_{pc}}$$

$$\frac{\iota(n) = Load \qquad \mu(p) = m@L_2}{\mu\ [p@L_1, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [m@(L_1 \vee L_2), \sigma]\ (n+1)@L_{pc}}$$

$$\frac{\iota(n) = Store \qquad \mu(p) = k@L_3 \qquad L_1 \vee L_{pc} \leq L_3 \qquad \mu(p) \leftarrow (m@L_1 \vee L_2 \vee L_{pc}) = \mu'}{\mu\ [p@L_1, m@L_2, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu'\ [\sigma]\ (n+1)@L_{pc}}$$

$$\frac{\iota(n) = Jump}{\mu\ [n'@L_1, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [\sigma]\ n'@(L_1 \vee L_{pc})}$$

$$\frac{\iota(n) = Bnz\ k \qquad n' = n + ((m = 0)?1 : k)}{\mu\ [m@L_1, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [\sigma]\ n'@(L_1 \vee L_{pc})}$$

$$\frac{\iota(n) = Call}{\mu\ [n'@L_1, a, \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [a, (n+1)@L_{pc}; \sigma]\ n'@(L_1 \vee L_{pc})}$$

$$\frac{\iota(n) = Ret}{\mu\ [n'@L_1; \sigma]\ n@L_{pc} \xrightarrow{\tau} \mu\ [\sigma]\ n'@L_1}$$

C Regeltabelle $\mathcal{R}^{abstract}$

opcode	*allow*	e_{rpc}	e_r
add	*true*	LAB_{pc}	$LAB_1 \sqcup LAB_2$
output	*true*	LAB_{pc}	$LAB_1 \sqcup LAB_{pc}$
push	*true*	LAB_{pc}	BOT
load	*true*	LAB_{pc}	$LAB_1 \sqcup LAB_2$
store	$LAB_1 \sqcup LAB_{pc} \sqsubseteq LAB_3$	LAB_{pc}	$LAB_1 \sqcup LAB_2 \sqcup LAB_{pc}$
jump	*true*	$LAB_1 \sqcup LAB_{pc}$	$--$
bnz	*true*	$LAB_1 \sqcup LAB_{pc}$	$--$
call	*true*	$LAB_1 \sqcup LAB_{pc}$	LAB_{pc}
ret	*true*	LAB_1	$--$

Ergänzung zu Kapitel 4.2: Die Ausdrücke der DSL enthalten zusätzlich die Verbandoperation $\sqsubseteq$ („can flow to“, vgl. Generator *genFlows*). *BOT* ist die symbolische Schreibweise für $\bot$. $--$ bedeutet, dass das Label des Ergebnisses irrelevant ist.

Literatur

[1] T. Hawkins. Redesigning the Computer for Security. `http://cufp.org/2013/slides/hawkins.pdf`, September 2013. Abgerufen am 22.04.2015.

[2] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A Verified Information-Flow Architecture. `http://www.crash-safe.org/assets/verified-ifc-popl2014.pdf`, January 2014. Abgerufen am 17.05.2015.

[3] C. Hriţcu. CRASH/SAFE: Clean-slate Co-design of a Secure Host Architecture. `http://prosecco.gforge.inria.fr/personal/hritcu/talks/crash-safe-darmstadt.pdf`, December 2012. Abgerufen am 01.06.2015.

[4] G. Sullivan. SAFE - A Clean-Slate, Secure Computing Platform. `http://www.mys5.org/Proceedings/2014/Day_3_S5_2014/2014-S5-Day3-05_Sullivan.pdf`, June 2014. Abgerufen am 02.06.2015.

[5] A. Thomas, J. McGrath, S. Chiricescu, S. Iyer, D. Wittenberg, B. Karel, N. Vasilakis, C. Hriţcu, J. M. Smith, B. C. Pierce, and A. DeHon. Towards a Zero-Kernel Operating System. `http://prosecco.gforge.inria.fr/personal/hritcu/publications/zkos_draft_jan10_2013.pdf`, October 2013. Abgerufen am 22.04.2015.

[6] B. C. Pierce. Verification Challenges of Pervasive Information Flow. `http://www.cis.upenn.edu/~bcpierce/papers/CRASH-PLPV-2012.pdf`, January 2012. Abgerufen am 22.04.2015.

[7] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. Exceptionally Available Dynamic IFC. `http://www.cs.ox.ac.uk/ralf.hinze/WG2.8/30/slides/catalin-paper.pdf`, July 2012. Abgerufen am 30.05.2015.

[8] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. `http://www.cse.chalmers.se/~andrei/csf10.pdf`, July 2010. Abgerufen am 30.05.2015.

[9] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 56–70, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] C. Hriţcu and K. Bhargavan. Speeding up Theorem Proving with Property-Based Testing. `http://prosecco.gforge.inria.fr/personal/hritcu/students/topics/2014/quick-chick.pdf`, 2014. Abgerufen am 28.04.2015.

[11] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A Verified Information-Flow Architecture for SAFE. `http://www.labri.fr/perso/casteran/LTP2013/Demange.pdf`, November 2013. Abgerufen am 06.06.2015.

[12] A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A Verified Information-Flow Architecture (Long version). `http://www.crash-safe.org/assets/verified-ifc-long-draft-2013-11-10.pdf`, November 2013. Abgerufen am 06.06.2015.

[13] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] C. Zhang. Unwinding Conditional Noninterference. `http://arxiv.org/pdf/1003.3893.pdf`, March 2010. Abgerufen am 25.05.2015.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, New York, NY, USA, 2009. ACM. Abgerufen am 01.06.2015.

[16] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MILS Architecture for High-Assurance Embedded Systems. `http://www.researchgate.net/profile/Paul_Oman/publication/220309643_The_MILS_architecture_for_high-assurance_embedded_systems/links/0912f50fee695f0273000000.pdf`, February 2005. Abgerufen am 02.06.2015.