Proof-Carrying Code

Stefan Berghofer

"Perlen der Weisheit"

Mobiler Code und Sicherheit

- Browser-Plugins
- Active-X Controls
- Java-Applets
- Gerätetreiber
- Software für Mobiltelefone, SmartCards
- ...

Gefahren

- Löschen / Verändern von Daten
- Ausspionieren / Mißbrauch von Daten
- Systemabstürze
- ...

$\Longrightarrow$ Lösungsmöglichkeiten?

Sicherheitskriterien (“Safety Policies”)

- Memory Safety
  - Kein Dereferenzieren ungültiger Pointer
  - Kein Zugriff auf unzulässige Speicherstellen
- Control-Flow Safety
  Sprünge nur zu zulässigen Adressen
- Type Safety
  Operationen finden nur auf Werten von geeignetem Typ statt.
- . . .

Sicherheit durch Authentifizierung

- “Vertrauenswürdige” Stellen dürfen Programme “signieren”
- Basiert auf Public-Key Kryptographie
- Ursprung des Programms identifizierbar
- Schutz vor Manipulation auf Übertragungsweg
- Beispiele: Microsofts *Authenticode*, signierte Active-X Controls

Probleme

- Signatur ist keine Garantie für Fehlerfreiheit
- Export- / Verwendungsbeschränkungen für Krypto-Software
- Wer darf Programme signieren? Wie werden Zertifikate vergeben?
- Aktueller Vorfall:

  > *Microsoft weist darauf hin, dass die Firma VeriSign an Unbekannte digitale Zertifikate vergeben hat, die es ermöglichen, Software-Code unberechtigterweise mit dem Namen “Microsoft Corporation” zu signieren. . . .*

  (siehe `http://www.microsoft.com/GERMANY/Support/virus.htm`)

Sicherheit durch Hardware-basierte Adressräume

- Speicherschutz unterstützt durch spezielle Hardware
- Management von Prozeßkontexten durch Betriebssystem

Probleme

- Komplexe Hardware und Betriebssysteme erforderlich – bei SmartCards und Embedded Systems u.U. nicht vorhanden
- Prozeßkontextwechsel teuer
- Nicht geeignet, wenn direkter Zugriff auf Hardware erforderlich (z.B. Gerätetreiber)
- Nicht für Echtzeitanwendungen geeignet ($\rightarrow$ RT-Linux)

Sicherheit durch Programmiersprachen-Semantik

- Typsysteme (ML, Java)
- Static checking (z.B. Java Bytecode Verifier)
- Dynamic checking
  - Interpretation (z.B. JVM)
  - JIT-Compiler, der Code für Laufzeit-Checks einfügt

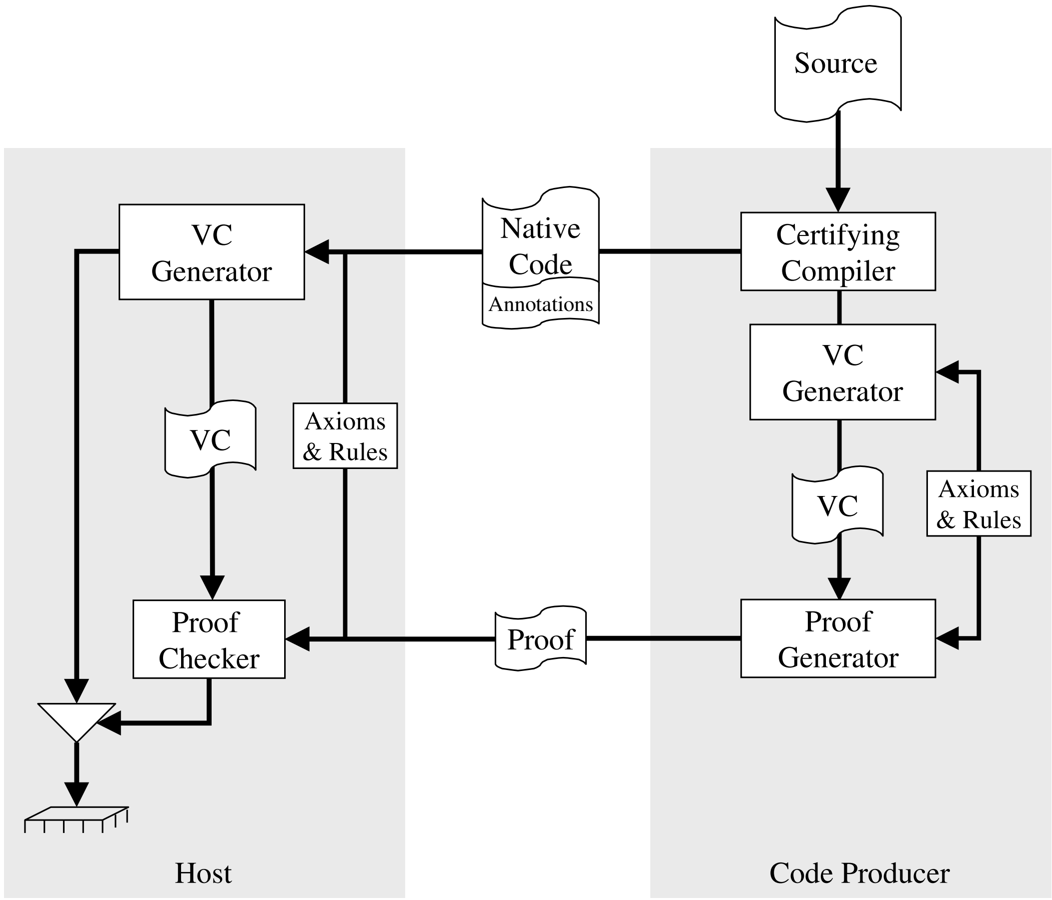
Probleme

- Typsysteme nicht immer ausdrucksstark genug
- Interpretation relativ langsam
- Vertrauen in Compiler erforderlich (große "trusted code base")

Alternative: Proof-Carrying Code

- G. Necula, *24th Annual Symposium on Principles of Programming Languages (POPL'97)*
- Code-Erzeuger liefert Programm + Beweis von (Sicherheits-)Eigenschaften
- Empfänger kann Beweis überprüfen
- Erfordert nur relativ unkomplizierte, vertrauenswürdige Infrastruktur (Proofchecker)
- Beweise prüfen ist einfacher als Beweise finden
- Alle logisch formalisierbaren Eigenschaften prüfbar

Architektur eines Proof-Carrying Code Systems



(Quelle: C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, K. Cline. A Certifying Compiler for Java. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'00)*)

Mögliche Manipulationsszenarien

- Code wurde verändert
  - VCs ändern sich nicht $\Rightarrow$ Beweis korrespondiert noch mit VCs $\ddot{\smile}$
  - VCs ändern sich $\Rightarrow$ Beweis korrespondiert nicht mehr mit VCs $\ddot{\frown}$
- Beweis wurde verändert
  - Beweis logisch gültig und korrespondiert mit VCs $\ddot{\smile}$
  - Beweis logisch ungültig oder korrespondiert nicht mit VCs $\ddot{\frown}$
- Beweis und Code wurden verändert
  - Beweis logisch gültig und korrespondiert mit (geänderten) VCs $\ddot{\smile}$
  - Beweis nicht logisch gültig oder korrespondiert nicht mit (geänderten) VCs $\ddot{\frown}$

Wichtige Fragen

- Was ist ein Beweis?
- Wann ist ein Beweis logisch gültig?
- Wie werden Verifikationsbedingungen dargestellt?
- Wie erzeugt man die Verifikationsbedingungen?
- Wie beweist man die Verifikationsbedingungen?

Logical Frameworks

- Allgemeines Werkzeug zur Repräsentation verschiedenster Logiken (Komponenten: logische Formeln, Terme, Beweise)
- Basieren auf getypten λ-Kalkülen:

$$\Gamma \vdash M : A \qquad \text{“}M \text{ hat den Typ } A\text{”}$$
$$\Gamma \vdash M : A \qquad \text{“}M \text{ ist ein Beweis der Formel } A\text{”}$$

- “Propositions as Types”, “Proofs as Terms”
- Beweisprüfung $=$ Typechecking
- Vertreter
  - Edinburgh LF (Harper, Honsell, Plotkin)
    Implementierung: Elf / Twelf (Pfenning, Schürmann)
  - Calculus of Constructions (Coquand, Huet)
    Implementierung: Coq
  - Isabelle (Paulson, Nipkow)

Repräsentation von Beweisen

Typechecking von LF-Termen

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{Var} \qquad\qquad \frac{c : A \in \Sigma}{\Gamma \vdash c : A}\ \text{Const}$$

$$\frac{\Gamma, x : A \vdash M \ : \ B}{\Gamma \vdash \lambda x : A.M \ : \ \Pi x : A.B}\ \text{Abs} \qquad \frac{\Gamma \vdash M \ : \ \Pi x : A.B \quad \Gamma \vdash N \ : \ A}{\Gamma \vdash M\ N \ : \ B[N/x]}\ \text{App}$$

Notation: $\quad \Pi x : A.B \ \approx \ A \to B \qquad$ falls $x \notin \text{FV}(B)$

Logiken als LF-Signaturen

```
andi   : ΠP:pred, Q:pred. pf P → pf Q → pf (& P Q).
andel  : ΠP:pred, Q:pred. pf (& P Q) → pf P.
ander  : ΠP:pred, Q:pred. pf (& P Q) → pf Q.
impi   : ΠP:pred, Q:pred. (pf P → pf Q) → pf (=> P Q).
```

```
λP:pred, Q:pred. impi (& P Q) (& Q P)
  (λH:& P Q. andi Q P (ander P Q H) (andel P Q H))
    : ΠP:pred, Q:pred. pf (=> (& P Q) (& Q P))
```

Kompression von Beweisen

- Beobachtung: Beweisterme enthalten viele Redundanzen (z.B. mehrfaches Vorkommen von "`& P Q`")
- Lösung: implizite Darstellung ("LF_i")

```
λP:_, Q:_. impi _ _
  (λH:_. andi _ _ (ander _ _ H) (andel _ _ H))
```

- Rekonstruktion fehlender Information ("_"):
  Lösen von Constraints zwischen Termen mittels *Unifikation*
- Achtung: Quantorenregeln wie

```
alle  : ΠP:exp → pred, E:exp. pf (all P) → pf (P E).
```

  erfordern *higher order* Unifikation
  - Allgemeine higher order Unifikation (G. Huet, 1974): unentscheidbar, keine eindeutigen allgemeinsten Unifikatoren, aber: semi-entscheidbar
  - Spezialfall: "Pattern Unifikation" (D. Miller, 1991, λ-Prolog): entscheidbar, eindeutige allgemeinste Unifikatoren
- Alternative: "Orakel-String" statt LF_i Term. Idee: Proof Checker = LP-Interpreter. "Orakel" codiert Entscheidungen bei Nichtdeterminismus (Wahl von Regeln / Unifikatoren) (G. Necula, S. Rahul, POPL 2000)

Safety Policies in LF

```
saferd   : ΠA:exp, T:exp, T':exp, L:exp, I:exp, RO:exp,
           ADDR:exp, M:exp, SIZET:exp, OFF:exp.
            pf (= ADDR (+ A I)) →
            pf (fieldOf T SIZET OFF T') →
            pf (arrIdx I L SIZET OFF) →
            pf (of A (array RO T L)) →
            pf (saferd M ADDR).

safewr   : ΠA:exp, T:exp, T':exp, L:exp, I:exp,
           ADDR:exp, M:exp, E:exp, SIZET:exp, OFF:exp.
            pf (of E T') →
            pf (= ADDR (+ A I)) →
            pf (fieldOf T SIZET OFF T') →
            pf (arrIdx I L SIZET OFF) →
            pf (of A (array rw T L)) →
            pf (safewr M ADDR E).
```

Generieren von Verifikationsbedingungen

- Ausgangspunkt: Annotierter Code
- Annotationen:
  - Schleifeninvarianten
  - Vor- und Nachbedingungen für Funktionen / Methoden
- Rechnen mit symbolischen Registerinhalten

  $Regs_F = \{r_1, \ldots, r_R, \texttt{mem}, l_1, \ldots, l_{Local_F}\}$

  $\sigma : Regs_F \to E$
- Registrierung bereits angetroffener Invarianten ($\mathcal{L}$)

Safe Assembly Language

$$
\begin{array}{lcl|l|l}
I & ::= & r \leftarrow r' & r \leftarrow n & r \leftarrow r' \; \texttt{EOP} \; r'' \\
 & | & \texttt{jump} \; n & \texttt{cond COP}(r), n & \\
 & | & \texttt{call} \; F & \texttt{ret} & \\
 & | & r \leftarrow M[r'] & M[r'] \leftarrow r & \\
 & | & \texttt{inv} \; P, Mod & \ldots &
\end{array}
$$

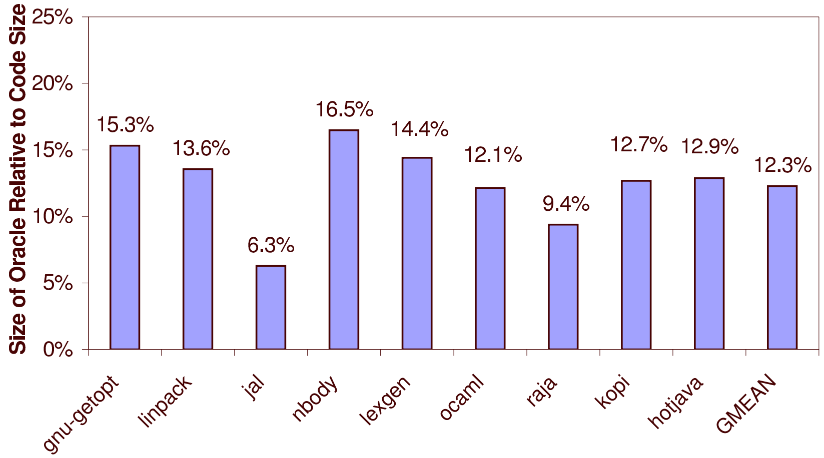
Generieren von Verifikationsbedingungen — Fortsetzung

F_i	Nebenbedingung	Verifikationsbedingung
$r \leftarrow r'$	$i+1 \in \mathit{Dom}(F)$	$\mathit{SE}(i+1, \sigma[r \mapsto \sigma(r')], \mathcal{L})$
$\mathtt{jump}\ n$	$n+i+1 \in \mathit{Dom}(F)$ $n < 0 \Rightarrow F_{n+i+1} = \mathtt{inv}\ \ldots$	$\mathit{SE}(n+i+1, \sigma, \mathcal{L})$
$\mathtt{cond}\ \mathtt{COP}(r), n$	$i+1 \in \mathit{Dom}(F)$ $n+i+1 \in \mathit{Dom}(F)$ $n < 0 \Rightarrow F_{n+i+1} = \mathtt{inv}\ \ldots$	$\mathtt{safecop}(\sigma(r)) \wedge$ $\mathtt{cop}(\sigma(r)) \Rightarrow \mathit{SE}(n+i+1, \sigma, \mathcal{L}) \wedge$ $\neg\mathtt{cop}(\sigma(r)) \Rightarrow \mathit{SE}(n+1, \sigma, \mathcal{L})$
$r \leftarrow M[r']$	$i+1 \in \mathit{Dom}(F)$	$\mathtt{saferd}(\sigma(\mathtt{mem}), \sigma(r')) \wedge$ $\mathit{SE}(i+1,$ $\quad \sigma[r \mapsto \mathtt{sel}(\sigma(\mathtt{mem}), \sigma(r'))], \mathcal{L})$
$M[r'] \leftarrow r$	$i+1 \in \mathit{Dom}(F)$	$\mathtt{safewr}(\sigma(\mathtt{mem}), \sigma(r'), \sigma(r)) \wedge$ $\mathit{SE}(i+1, \sigma[\mathtt{mem} \mapsto$ $\quad \mathtt{upd}(\sigma(\mathtt{mem}), \sigma(r'), \sigma(r))], \mathcal{L})$
$\mathtt{inv}\ P, \{r_1, \ldots, r_k\}$	$i \notin \mathit{Dom}(\mathcal{L})$ $i+1 \in \mathit{Dom}(F)$	$\sigma(P) \wedge \forall y_1 \ldots y_k.\ \sigma'(P) \Rightarrow$ $\mathit{SE}(i+1, \sigma', \mathcal{L} + (i, \sigma'))$ wobei $y_1, \ldots, y_k$ neue Variable $\sigma' = \sigma[r_1 \mapsto y_1, \ldots, r_k \mapsto y_k]$
$\mathtt{inv}\ P, \mathit{Mod}$	$i \in \mathit{Dom}(\mathcal{L})$ $i+1 \in \mathit{Dom}(F)$	$\sigma(P) \wedge$ $\forall r \in \mathit{Regs}_F \setminus \mathit{Mod}.\ \sigma(r) = \mathcal{L}\ i\ r$

Praktische Anwendbarkeit

- Spin-Off der CMU: Cedilla Systems (`http://www.cedillasys.com/`)
- Implementierung: SpecialJ Compiler für Java

Programm	Codegröße (bytes)	LF_i Größe	Prüfung LF_i (ms)	Orakelgröße	LP-Interpr.
jal	27080	53328	84	1698	314
lexgen	109196	413538	655	15726	1948
raja	126364	371276	747	11854	2030
hotjava	2747548	10813894	19757	354034	53491



(Quelle: G. C. Necula. A Scalable Architecture for Proof-Carrying Code. In *Fifth International Symposium of Functional and Logic Programming (FLOPS 2001)*)

Verwandte Arbeiten

- Result Checking (LEDA): Algorithem liefern Zusatzinformation zur einfachen Überprüfung des Ergebnisses, z.B. Planaritätstest:
  - Planare Einbettung, falls Graph planar
  - Kuratowski-Teilgraph, falls Graph nicht planar
- Lightweight Bytecode Verification (E. Rose, K. Rose / G. Klein, T. Nipkow)
  Bytecode Verification $\approx$ Typinferenz
  Annotation von Sprungzielen mit Typen $\Rightarrow$ linearer Zeit- / Platzbedarf
- Programmextraktion aus konstruktiven Beweisen
  G. Huet, A. Chander: "Prooflets"

Programmextraktion aus konstruktiven Beweisen

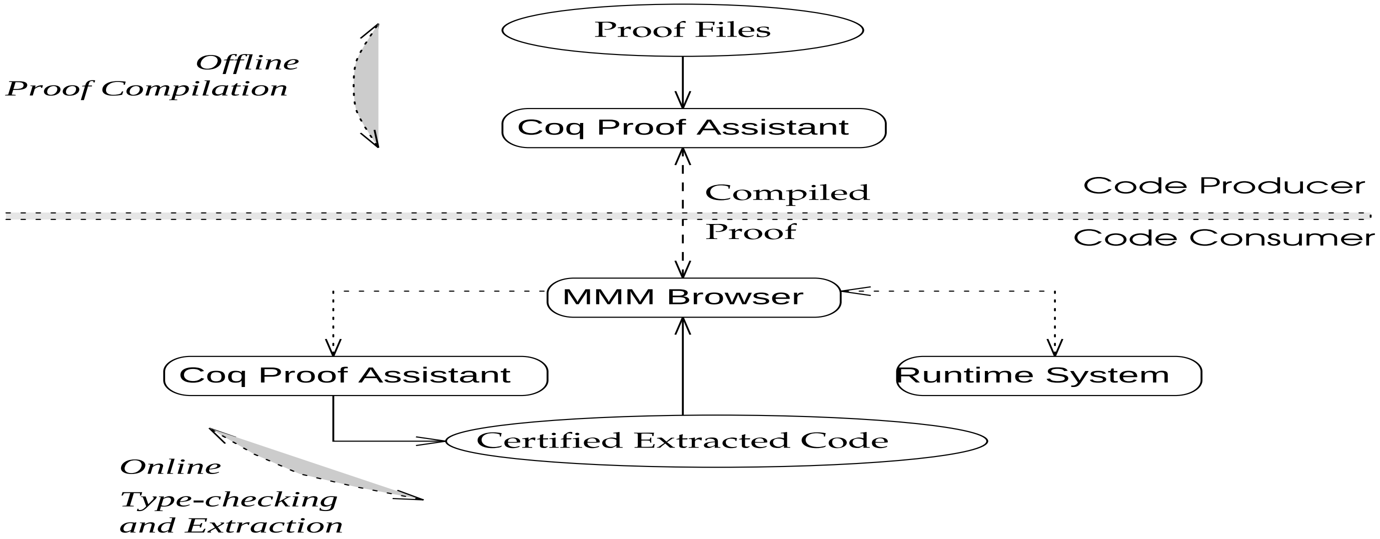
- Idee: $\mathit{Konstruktiver}$ Beweis von

  $$\forall x_1 : \sigma_1 \ldots x_m : \sigma_m.\ \exists y_1 : \tau_1 \ldots y_n : \tau_n.\ \varphi\ x_1\ \ldots\ x_m\ y_1\ \ldots\ y_n$$

  beinhaltet Programm vom Typ

  $$\sigma_1 \to \cdots \to \sigma_m \to \tau_1 \times \cdots \times \tau_n$$

- Extrahiertes Programm erfüllt Spezifikation φ (*“Realisierbarkeit”*)
- Beispiel: Beweis von $\forall a\ b.\ \exists r\ q.\ a = (b+1) \cdot q + r \wedge r \leq b$ enthält Divisionsalgorithmus
- Implementierung: G. Huet, A. Chander: “Prooflets”



Links

- George Necula's Proof-Carrying Code Page
  `http://www.cs.berkeley.edu/~necula/pcc.html`
- Proof-Carrying Code in Princeton (A. Appel)
  `http://www.cs.princeton.edu/sip/projects/pcc/`
- Peter Lee's Proof-Carrying Code Page
  `http://www.cs.cmu.edu/~petel/papers/pcc/pcc.html`
- CMU Fox Project
  `http://foxnet.cs.cmu.edu/`