Engineering Formal Security Policies for Proof-Carrying Code

Andrew Bernard
Carnegie Mellon University
Thesis Committee

- Peter Lee, chair
- Karl Crary
- Frank Pfenning
- Fred B. Schneider, Cornell University
Code Safety

- We use an increasing number of programs
- The number of program sources is also increasing
- But are these programs safe for us to run?

- Technologies for *code safety* enable us to say “yes” before we run a program on our computer
Challenges to Code Safety

- Byte-code interpreters are slow
  - Unacceptable battery drain for small devices
- Just-in-time compilers are large and complex
  - Critical bugs can be expensive to fix
- Digitally-signed programs aren’t trustworthy
  - Signature doesn’t ensure that the code is actually safe
  - Everyone is vulnerable if the private key is compromised (e.g., Microsoft)
The Right Architecture: Proof-Carrying Code (PCC)

- Good performance
  - Compile in advance with all optimizations
  - Fewer run-time checks
- Small trusted computing base
  - Possible to verify informally
- No trusted third parties
  - Each host trusts only itself
A code producer sends a program and its safety proof to a code consumer
- A certifying compiler constructs the program and proof
  - A proof checker checks them against a security policy
PCC: Principles

- **Code Producer**
  - Certifying Compiler
  - Program
  - Proof

- **Code Consumer**
  - Proof Checker
  - Security Policy

Untrusted

**Certification** can be complex

**Enforcement** should be simple

OS Vendor

No third party to “vouch” for safety
PCC: Practice

• SpecialJ Certifying Compiler [Colby, et al. 00]
  – Developed by Cedilla Systems
  – Certified, optimized x86 machine code from Java source code (no byte-code interpreter)
  – Safety policy is Java type safety
  – Translation to object code preserves type safety
  – Scales to large programs (e.g., JDK 1.3, HotJava)

• LF Logical Framework [Harper, et al. 87]
  – Flexible internal language for propositions and proofs
  – Proof checking is type checking
PCC: Reality

**Program Proof Code Producer**

SpecialJ

**Certifying Compiler**

Loop Invariants Procedure Specifications (for safety properties)

**Annotations**

**Proof**

**Machine Semantics**

General Security Policy

**VC Generator**

**Code Consumer**

Proof Checker Security Policy
Verification-Condition Generator

- Verification condition (VC) is true only if program is safe (but not necessarily correct)
- Derived by *symbolic evaluation* [Necula/Lee 97]
  - Simulates program operations on abstract state
- Proofs are scalable, but the VC generator is
  - complex: 16,000 lines of dense C
  - machine specific
  - compiler specific
  - source-language specific
  - security-policy specific
How is the Security Policy Represented?

• A combination of lots of C code and typing rules
  – Type systems are relatively trustworthy
  – But C code is more obscure and error prone
What if we want …

• more than Java type safety?
  – *e.g.*, resource bounds, information flow

• to manage security policies?

• to manipulate security policies?

• **We need to change the VC generator!**
  – A better solution: a *universal* enforcement mechanism
  – Separate *policy* from *mechanism*
A Formal Language for Security Policies

- Formal security policies for a universal checker
  - Security policy can be part of the certificate
- Temporal logic is an attractive policy notation
  - Direct specifications: $\Box (pc < 1000)$
  - Well-understood semantics
  - Can express a wide variety of security properties
  - Can reuse existing type-safety specifications
  [Necula/Lee 97]
The Goal

- To achieve PCC by
  - Proving directly that a program satisfies a formal security policy
  - Instead of generating and proving an intermediate VC
- No VC generator
- Key question: is it practical?

16k lines of C
The Approach: Verifiable Logic Programs

- Certificate is a *program* for generating a proof
  - Extracts and proves its own VCs
  - Sound by construction
- Drawback: proof checking is slower (so far)
  - Proving more than SpecialJ safety proofs
- Possible “next step” for PCC proof checking?
  - LF type reconstruction [Necula/Lee 96]
  - Oracle-based theorem proving [Necula/Rahul 01]
  - Verifiable logic programs
Thesis Statement

• It is practical to engineer a system for proof-carrying code in which policy is separated from mechanism.

• In particular, I examine a generic implementation of the PCC infrastructure that accepts a wide variety of security properties encoded in a formal specification language.
The Rest of this Talk

- Framework
- Code Consumer
- Code Producer
- Proof Engineering

Certifying Compiler

Program

Proof

Proof Checker

Security Policy
Framework

Temporal Logic

Machine Model
Temporal Logic

- Truth is relative to a specific *time*
  - Propositions hold over finite or infinite intervals
- Excellent representation for security properties
  - *How* the program computes a result
Temporal Logic Syntax

• Linear-time 1st-order temporal logic [Manna/Pnueli 80]
  – Identify time with CPU clock

• Expressions $e ::= a \mid x \mid c \mid f(e_1, \ldots, e_k)$
  – Parameters $a$ refer to the machine state (e.g., $pc$)

• Propositions $p ::= R(e_1, \ldots, e_k)$
  | $p_1 \land p_2$ | $p_1 \lor p_2$ | $p_1 \supset p_2$
  | $\forall x. \ p_1$ | $\exists x. \ p_1$ | $\bigcirc p_1$ | $\square p_1$ | $p_1 \cup p_2$
Temporal Logic as a Security-Policy Language

• Combining security policies
  – Conjunction: $p_1 \land p_2$
  – Disjunction: $p_1 \lor p_2$

• Tracking execution history (security automata)
  – History parameters: $\square (q = e \supset \bigcirc (q = e')) \supset \square p(q)$

• Modular security policies
  – Private histories: $\forall y. \square (y = e \supset \bigcirc (y = e')) \supset \square p(y)$
Abstract Machine Model

- Simplified machine model for this talk
- Three parameters for machine state
  - \texttt{pc}: program counter
  - \texttt{g}: general-purpose register file
  - \texttt{m}: memory
- Instruction set is unimportant
Code Consumer

Certifying Compiler

Program

Proof

Proof Checker

Security Policy
Formal Machine Semantics

• Provides a basis for proof checking
  – Security policy must follow from machine semantics
• The transition relation
  – Effect of an instruction on a state [Pnueli 77]
  – Syntactic rather than semantic (e.g., model checking)
• Theorem: soundness
  – Follows directly from operational semantics
Proof Checking

- Enforces *all* temporal-logic security properties
  - All safety properties (*e.g.*, memory safety, resource bounds, access control, Java security manager)
  - Most familiar liveness properties (*e.g.*, termination)
  - Can’t express noninterference (*e.g.*, information flow)
- Proof checker has a logic-program interpreter
  - Reconstrucsts omitted proof fragments
- Handwritten proofs are possible, but…
Code Producer

Certifying Compiler

Program

Proof

Proof Checker

Security Policy
• Proof construction is harder than proof checking
  – We can enforce more properties than we can certify
    • This may be inevitable
• Decouple enforcement from certification
  – Many approaches to certification
  – Get enforcement right “once and for all”
• Focus on certifying type safety
  – Needed for many applications
  – First test case for automatic certification
Automatic Proof Construction

- Build an *adapter* for the SpecialJ compiler
  - Automatic proofs of type safety
  - Experiment with larger examples
- Use a *logic of programs* for safety properties
  - Derive from temporal logic [Gordon 89]
  - Instantiate for SpecialJ type safety
- How does SpecialJ work?
SpecialJ Symbolic Evaluation

- Interpret program using machine semantics
  - Symbolic machine state is a formal expression
  - Unknown values are variables
  - Based on an implicit program logic
- Emit proof obligations for dangerous instructions
- Handle loops using *recurring loop invariants*
  - Each invariant leads to another invariant, and we're safe in the meantime
Proof Generation

- Generate a “skeleton” of program-logic rules that simulate the SpecialJ symbolic evaluator
  - Safety proofs from SpecialJ discharge premises of program-logic rules
- The code producer supplies the untrusted program logic [Appel/Felty 00]
Program Logic

Certifying Compiler

Program

Proof

Proof Checker

Security Policy
A Logic of Programs for Invariance Properties

- A specialized logic for reasoning about programs
  - Proves invariance in addition to partial correctness
  - Verifies each procedure independently
- Shows that an invariance property holds until a specific goal property is reached
  - Goal is initially a procedure postcondition
- An invariance property is a property of individual machine states
Conventions

- $e$ is a symbolic machine-state tuple $(e_{pc}, e_g, e_m)$
  - Parameters for unknown values
  - Example: $(25, \text{upd}(a_g, r_0, 5), a_m)$
- $s$ is a parameter always equal to $(pc, g, m)$
- $p_{\text{safe}}$ is an invariance property (e.g., type safety)
- $p_{\text{g1}}$ is the current goal property
Specifications

- Specifications on $s$ (no temporal operators)
  - $p_i$ is a loop invariant
  - $p_p$ is a procedure precondition
  - $p_q$ is a procedure postcondition
  - $x_0$ is free: instantiated with reference machine state

- Example loop invariant (for code address 25):
  $$\pi_{pc}(s) = 25 \land r_0(\pi_g(s)) : \text{int} \land \pi_m(s) = \pi_m(x_0)$$
  $$pc = 25 \land r_0(g) : \text{int} \land m = \pi_m(x_0)$$
Judgments

- **Transition**
  - The successor of state \( e \) is \( e' \)

- **Evaluation**
  - From state \( e \), \( p_{safe} \) holds until \( p_{gl} \) holds

- **Strict Evaluation**
  - \( p_{safe} \) must also hold for at least one step

- **Procedure Call**
  - Once \( p_p \) holds, \( p_{safe} \) holds until \( p_q \) holds
  - Derive this for initial entry point
Strict Evaluation Rules

\[
\begin{align*}
\vdash [e/s] p_{\text{safe}} & \quad \vdash e \rightarrow e' & \quad \vdash e' \xrightarrow{p_{\text{safe}}} p_{g_{\downarrow}} & \xrightarrow{\rightarrow} + i_{\downarrow} \\
\vdash e \xrightarrow{p_{\text{safe}}} + p_{g_{\downarrow}} & \\
\vdash e \xrightarrow{p_{g_{\downarrow}}} + e & \xrightarrow{\rightarrow} + e \\
\vdash e \xrightarrow{p_{g_{\downarrow}}} + p_{g_{\downarrow}} & \\
\vdash e \xrightarrow{p_{\text{safe}}} + p_{g_{\downarrow}} & \xrightarrow{\rightarrow} + e
\end{align*}
\]
Evaluation Rules

\[
\begin{align*}
\vdash [e/s] p_g & \quad \leadsto i_0 \\
\vdash p_{\text{safe}} & \quad \leadsto p_g \\
\vdash e & \quad \leadsto p_g \\
\vdash [e/x_0] [a/s] p_i & \quad \leadsto [e/x_0] p_i \lor p_g \\
\vdash p_{\text{safe}} & \quad \leadsto \alpha \lor [e/x_0] p_i \lor p_g & \quad \leadsto \text{loop}^\alpha, u \\
\vdash e & \quad \leadsto p_g
\end{align*}
\]
Proof Representation

<table>
<thead>
<tr>
<th>Decoding</th>
<th>Prelude</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>[96] P1 P2 ==&gt; P1 and P2.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| prl/sk_safe: ...
  = [d_r] [d_s] ...
| 6e 06 90 0d 0d 8f 89 0e
  60 78 60 60 7c 34 23 7d |

- Minimize total proof size for large programs
  - *Decoding* specifies binary-to-LF translation
  - *Prelude* provides derived rules
    - Includes the derived program logic, logic program
  - *Body* is a binary proof encoding
Proof Reconstruction

- An explicit proof is too large, even in binary
- Use the logic interpreter: reconstruct most of the proof on demand
  - Omit decidable fragments entirely
  - Map undecidable fragments onto minimal outlines
    - Explicitly constrain possible clauses [Pfenning 01]
    - Resembles “oracle” checking [Necula/Rahul 01]
  - Code producer chooses which parts to omit
Verifiable Logic Programs

• Only search over derived rules
  – Each derived rule has an explicit proof in the prelude
  – Code producer writes the logic program
    • Based on program logic
  – Optimize for the certification strategy
Experimental Results
Proof Size

- Doesn’t include prelude or decoding
Relative Proof Size

![Graph showing the relative proof size with Object-Code Size (bytes) on the x-axis and Proof Size (bytes) on the y-axis. The graph compares three different methods: LF, Oracle, and another method without a clear label. The LF method shows a sharp increase in proof size compared to the other two methods.]
Proof-Checking Time

- Measured on a 1.6GHz Athlon PC
Related Work
Foundational PCC

- *Foundational PCC* [Appel/Felty 00; Hamid, *et al.* 02] reconstructs PCC on higher-order logic
  - No trusted type system: derive in higher-order logic
- I want explicit security policies
  - Work with an existing compiler and safety proofs
  - Attack VC generator: less trustworthy than type system
    - 16,000 lines of C vs. 200 lines of LF
- *Foundational typed assembly language* [Crary 03; Crary/Sarkar 04]
Expressive Security Policies

- **Proof-carrying Code**
  - Resource bounds [Necula/Lee 98]

- **Typed Assembly Language (TAL)**
  - Security automata [Walker 00]
  - Capabilities [Crary, et al. 99]
  - Resource bounds [Crary/Weirich 00]
  - TALT-R [Vanderwaart/Crary 04]

- **Software Fault Isolation**
  - Security automata [Erlingsson/Schneider 99]
  - Edit automata [Walker 02]
Future Work

• Speed up proof checking
  – Where is the “sweet spot?”

• Automatic certification for more safety properties
  – Advanced type systems
  – Instrumentation

• Temporal logic is a particular choice of notation
  – Measure effect of other choices
Automatic Certification: Advanced Type Systems

- Type systems can check many safety properties
- Programmer provides the proof
  - Source code must be written such that it type checks
  - A typing derivation is a proof of safety
- Two approaches for PCC
  - Code consumer adopts another type system (easier)
  - Map typing derivations onto derived rules (harder)
Automatic Certification: Instrumentation

- Inline reference monitors (IRM) [Erlingsson/ Schneider 99]
  - Security automaton threaded through program
  - Run-time checks ensure program is safe
  - Tools exist to instrument Java bytecode (SASI, Naccio, Polymer)

- Code producer can also do this
  - Straightforward loop invariants and proofs
  - No IRM tool in the TCB
Conclusion

• Contributions
  – Enforcement for Temporal-Logic Properties
  – A Derived Program Logic for Safety Properties
  – Proof Engineering for Foundational Proofs
  – A Temporal-Logic Framework for PCC
  – A Foundation for SpecialJ

• Thanks: Michael Donohue, Stephen Magill