Assuring Software Protection in Virtual Machines

Andrew W. Appel

Princeton University
Software system built from components

Less-trusted components

(less-trusted functions)

INTERFACE

INTERFACE

INTERFACE

More-trusted components

Access Control

(core functions)

How to guarantee “no bypassing the interface”? 
Software system built from components

How to guarantee “no bypassing the interface”?  

1. Virtual Memory Protection

- Surprisingly complex implementation: hard to validate
- Interfaces are not very expressive: no fine-grained field-by-field access control.
- Method-calls must be cross-address space: slow!

- Time-tested and well understood
Software system built from components

How to guarantee “no bypassing the interface”?

2. Language-based security (e.g., Java-style type-checking)
   ☺ Fine-grain access-control possible
   ☻ Very expressive policy languages possible
   ☺ Specified in terms the programmer understands
   ☹ Complicated to implement securely
**Java or .Net virtual machine**

**APPLICATION DEVELOPER**
Java, C#, or VB
Source Program

Java (etc.)
compiler

Java or .Net byte codes

**USER’s MACHINE**

checker
JIT (just-in-time) Compiler

Machine-language program

Library 1  Library 2  Library 3

Java VM

Trusted Computing Base
Outline of talk

• Securing the JIT compiler
• Securing the libraries
• Securing the application
1. How to attack the JIT!

1. Find a bug in the JIT (every user has a copy!)

2. Write byte-code that tickles the bug (i.e., passes the checker but generates machine code that “cheats”)
Solution: prove a soundness theorem!

**Theorem:** If the *checker* accepts a byte-code program, then the *machine-language* program will respect its interfaces.

**Proof:** ?
How did you avoid all our pitfalls? Ha! That can’t possibly work!

Social Processes and Proofs of Theorems and Programs, by Richard DeMillo, Richard Lipton, and Alan Perlis,

POPL ’77

Abstract:
It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore...
Summary: why ‘verification’ must fail

• Specification of correct behavior impossibly complex
• Domain knowledge required
• Programmers not trained in logic & proof
• Proofs too huge and complex to produce
• Proofs too complex to check
• Prove source program, execute machine-language
  ▪ Semantics of source language ill-defined
  ▪ Compiler might have bugs

• Large programs can never be perfect
• Formal “machine” proofs don’t communicate ideas to humans
<table>
<thead>
<tr>
<th>People kept at it, though...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type Systems Research</strong></td>
</tr>
<tr>
<td>1978-2004</td>
</tr>
<tr>
<td>Provably* sound type systems; “Well-typed programs can’t go wrong.”</td>
</tr>
<tr>
<td>*in the mathematician’s sense</td>
</tr>
<tr>
<td>ML, Modula-3, Java...</td>
</tr>
<tr>
<td>(each one an advance over its successors!)</td>
</tr>
<tr>
<td><strong>Mechanical Proof</strong></td>
</tr>
<tr>
<td>1978-2004</td>
</tr>
<tr>
<td>(not “program verification”)</td>
</tr>
<tr>
<td>but proof-development and proof-checking tools</td>
</tr>
<tr>
<td>Boyer-Moore, HOL, Isabelle, NuPrl, Coq, ...</td>
</tr>
</tbody>
</table>
Typed Assembly Languages / PCC

(Necula '97; Morrisett '98)

Typed IL 1

Typed IL 2

Typed IL 3

Typed Assembly Lang.

Compiler Phase 1

Compiler Phase 2

Compiler Phase 3

Compiler Phase 4

Source Program

Code Producer

Code Consumer

Typechecker

Assembler

Native code

Execute

OK

TAL Code

L₁: r₃: τ₁
load r₃, 4(r₂)
add r₂, r₄, r₁

L₂: r₃: τ₁, r₁: int
store r₁, 4(r₇)
add r₇, 0, r₃
add r₇, 8, r₇

Trusted Computing Base

(Necula ‘97; Morrisett ‘98)
Java or .Net compatible

How big and complicated is the part that you have to trust?
Size of Trusted Computing Base

- Kaffe
- BulletTrain
- Chase, Hoover, Zadeck
- SpecialJ

- Open-source JVM, non-optimizing JIT
- Highly optimizing Java Compiler
- PCC system, optimizing compiler

- Untrusted Compiler
- Trusted Compiler (or checker)
- Core Runtime
- Memory Mgmt

Necula, Lee, et al.
Princeton FPCC compiler *(to scale!)*

![Image of Juan Chen]

**Juan Chen**

A Low-Level Typed Assembly Language with a Machine-checkable Soundness Proof

PhD Thesis, 2004

- Pre-existing components
- Custom components

---

**Standard ML of New Jersey front end**

**Type-preserving middle**

**MLRISC retargetable code generator**

- TAL
- Sparc code

**Checker**

- Execute

170k lines of code
What’s in the checker

(4000+1000 lines of code)

- 196 Sparc instructions
- 263 decoding rules
- 79 coercion operators+rules
- 48 substitution ops+rules
- 79 regmap, labelmap ops+rules
- 27 type operators
- 69 type refinement rules
- 98 wellformedness rules
- 51 expression operators
- 54 expression rules

Standard ML of New Jersey front end

Type-preserving middle

Retargetable code generator

Proof outline
Sparc code
Checker
Execute
Here’s one rule (can you trust this?)

(4000+1000 lines of code)

196 Sparc instructions
263 decoding rules
79 coercion operators+rules
48 substitution ops+rules
79 regmap, labelmap ops+rules
27 type operators
69 type refinement rules
98 wellformedness rules
51 expression operators
54 expression rules

regbind $\Gamma$ At A
realreg At Ar
regbind_val $\Gamma$ B Bt
realreg Bt Br
match_reg_or_imm $\Gamma$ C Cx
valueTy $\Gamma$ $\Phi$ N B int32
valueTy $\Gamma$ $\Phi$ N C int32
venv_add\ $\Gamma$ A int32 $\Phi$ $\Phi'$
decode_list L L' P P' (i_ADD Br Cx Ar)

$\Gamma \Rightarrow L M Q N P \Phi$

(e_prim A (p_arith a_add B C))

L' M Q N P' $\Phi'$
Machine-checkable proof (to scale!)

- Machine-checkable soundness proof: 130k lines of proof
- Standard ML of New Jersey front end
  - Type-preserving middle
  - Retargetable code generator
    - Checker
      - TAL
      - Sparc code
      - Execute

Axioms & Sparc spec. → Proof checker
Size of Trusted Code Base: 3028 lines

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic</td>
<td>135</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>160</td>
</tr>
<tr>
<td>Machine Syntax</td>
<td>460</td>
</tr>
<tr>
<td>Machine Semantics</td>
<td>1005</td>
</tr>
<tr>
<td>Safety Predicate</td>
<td>105</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1865</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/Output</td>
<td>43</td>
</tr>
<tr>
<td>Parser</td>
<td>428</td>
</tr>
<tr>
<td>DAG creation</td>
<td>111</td>
</tr>
<tr>
<td>Type checking</td>
<td>167</td>
</tr>
<tr>
<td>Interpreter</td>
<td>360</td>
</tr>
<tr>
<td>Proof checker</td>
<td>14</td>
</tr>
<tr>
<td>DAG manipulation</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1163</strong></td>
</tr>
</tbody>
</table>

Policy & Specification
- Safety Predicate: 6%
- Logic: 7%
- Arithmetic: 9%
- Machine Syntax: 25%
- Machine Semantics: 53%
- Safety Predicate: 6%

General-purpose
- Proof checker ("Flit")
  - Main program: 5%
  - Input/Output: 4%
  - Interpreter: 21%
  - Proof checker: 14%
  - DAG manipulation: 10%
  - Parser: 36%
Smallest possible Trusted Base

- Untrusted Compiler
- Trusted Compiler (or checker)
- Core Runtime
- Memory Mgmt

Bar chart showing the sizes of different systems in thousands of lines:
- Open-source JVM, non-optimizing JIT
- Highly optimizing Java Compiler
- PCC system, optimizing compiler
- Princeton FPCC Prototype

- Kaffe
- BulletTrain
- SpecialJ
- FPCC
Write JIT in type-safe VM language!

Code Producer
Source Program ↓
 Compiler Phase 1
 Java Bytecodes

JVM

Compiler Phase 2
Compiler Phase 3
Compiler Phase 4
Assembler
Typechecker

OK
Execute

Typechecker catches bugs in compilation, but can’t catch compiler trashing the VM!

Jikes gets this right; most others get it wrong.
2. How to attack the libraries!

1. Find a bug in the libraries!
2. Exploit...
Solution

Write libraries in the type-safe source language.

Not always possible!

Doesn’t always guarantee security!
Three categories of library code

<table>
<thead>
<tr>
<th>TYPE-UNSAFE</th>
<th>TYPE-SAFE SECURITY-RELEVANT</th>
<th>TYPE-SAFE USER LIBRARIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native-code libraries to support APIs not implementable in type-safe source language</td>
<td>Implemented in type-safe source language, provide sensitive capabilities such as I/O or security primitives</td>
<td>Not called by security-relevant libraries; utilities such as quicksort or GUI</td>
</tr>
</tbody>
</table>

(Example: JNI)
Snapshots of some JVMs *(to scale!)*

- **Sun Java 2 SDK 1.3**
  - JIT
  - Safe Libraries
  - Unsafe Libs.
  - RunT
  - GC

- **Hotspot (Java 2 SDK 1.3)**
  - JIT
  - Safe Libraries
  - Unsafe Libs.
  - RunT
  - GC

- **Kaffe (circa 2001)**
  - JIT
  - Safe Libraries
  - Unsafe Libs.
  - RunT

- **BulletTrain**
  - JIT
  - Safe Libraries
  - Unsafe Libs.

- **SpecialJ / Ginseng**
  - JIT
  - Safe Libraries
  - Unsafe Libs.
  - RunT
  - checker

- **IBM Tokyo/Toronto JVM**
  - JIT
3. Attack the application!

- How easily can the application programmer write secure and robust programs?

- Case study: Java
McGraw & Felten’s 12 rules

1. Don’t depend on initialization
2. Limit access to your classes, methods, & vars.
3. Make everything final, unless there’s a good reason not to
4. Don’t depend on package scope
5. Don’t use inner classes
6. Avoid signing your code
7. If you must sign your code, put it all in one archive file
8. Make your classes uncloneable
9. Make your classes unserializable
10. Make your classes undeserializable
11. Don’t compare classes by name
12. Secrets stored in your code won’t protect you
13. Beware reflection!
2. Limit access to your classes, methods, & vars.

3. Make everything final, unless there’s a good reason not to

12. Secrets stored in your code won’t protect you
It’s a shame these rules are needed...

1. Don’t depend on initialization
2. Limit access to your classes, methods, & vars.
3. Make everything final, unless there’s a good reason not to
4. Don’t depend on package scope
5. Don’t use inner classes
6. Avoid signing your code
7. If you must sign your code, put it all in one archive file
8. Make your classes uncloneable
9. Make your classes unserializable
10. Make your classes undeserializable
11. Don’t compare classes by name
12. Secrets stored in your code won’t protect you
13. Beware reflection!
Moral:

- Clean language design can make it easier to secure application code
Conclusion 1: JIT compiler

• Write JIT in the type-safe VM language
  (so executing the compiler won’t be exploitable)

• Use typed intermediate languages in JIT
  (so bugs in JIT can’t produce unsafe code)

• Note: conversion to typed ILs can be done incrementally, from the top down
Incremental development of typed ILs

Code Producer
Source Program ↓ Compiler Phase 1

VM
Compiler Phase 2
Compiler Phase 3
Compiler Phase 4
Assembler

↓ OK
Execute
• If you use typed intermediate languages today, we “verification weenies” can help you take advantage of that tomorrow.

• Buzzword: “Foundational Proof-Carrying Code”
Conclusion 2: libraries

- **Libraries in unsafe implementation language:**
  sometimes unavoidable, but minimize them to a tiny core

- **Privilege-granting libraries in safe VM language:**
  still better than doing them in C or C++

- **Safe libraries that are privilege-irrelevant:**
  these are what you want
Conclusion 3: application programmer

- If you need so many rules about what to avoid, then maybe something is wrong.