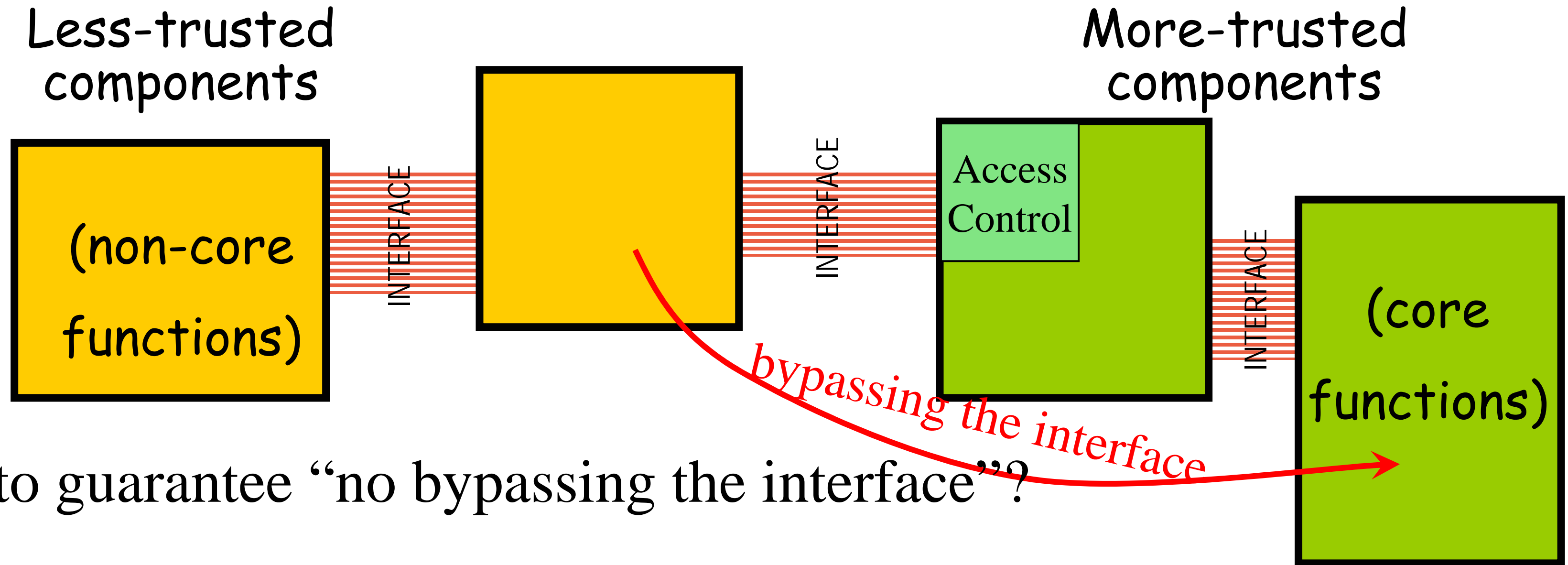
Assuring Software Protection in Virtual Machines

Andrew W. Appel

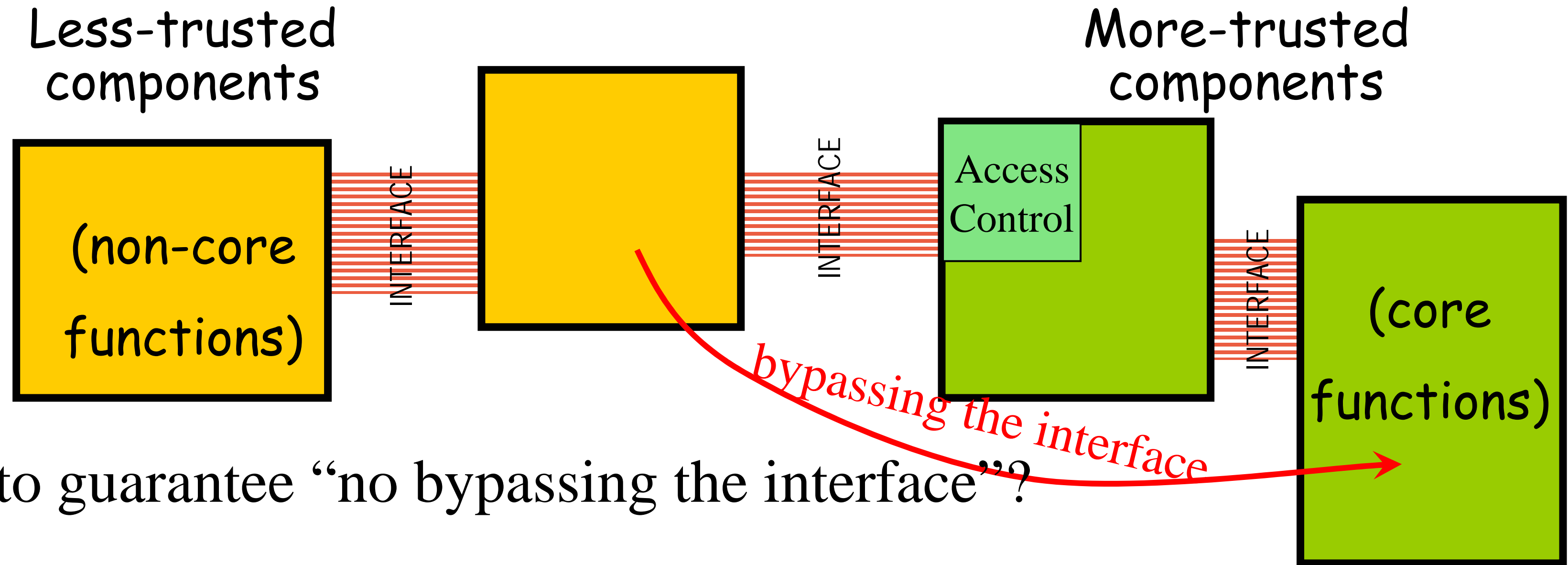
Princeton University

Software system built from components



How to guarantee “no bypassing the interface”?

Software system built from components



How to guarantee "no bypassing the interface"?

1. Virtual Memory Protection

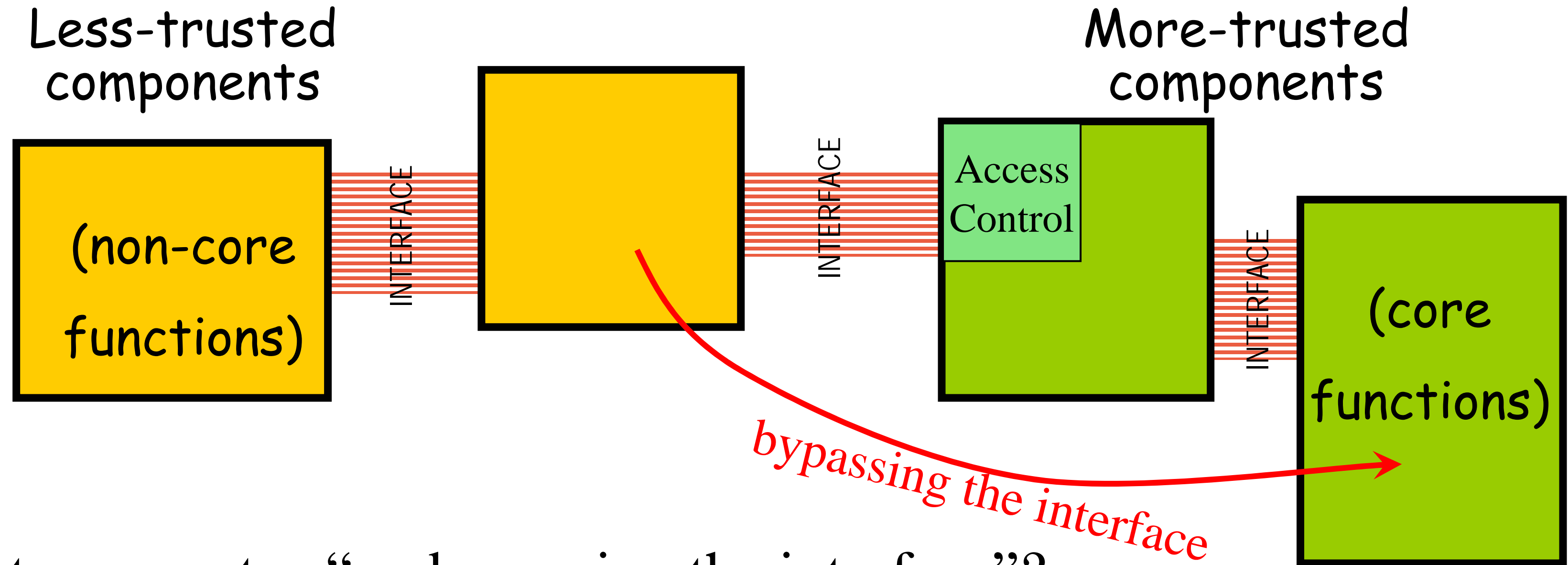
☹ Surprisingly complex implementation: hard to validate

☹ 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)

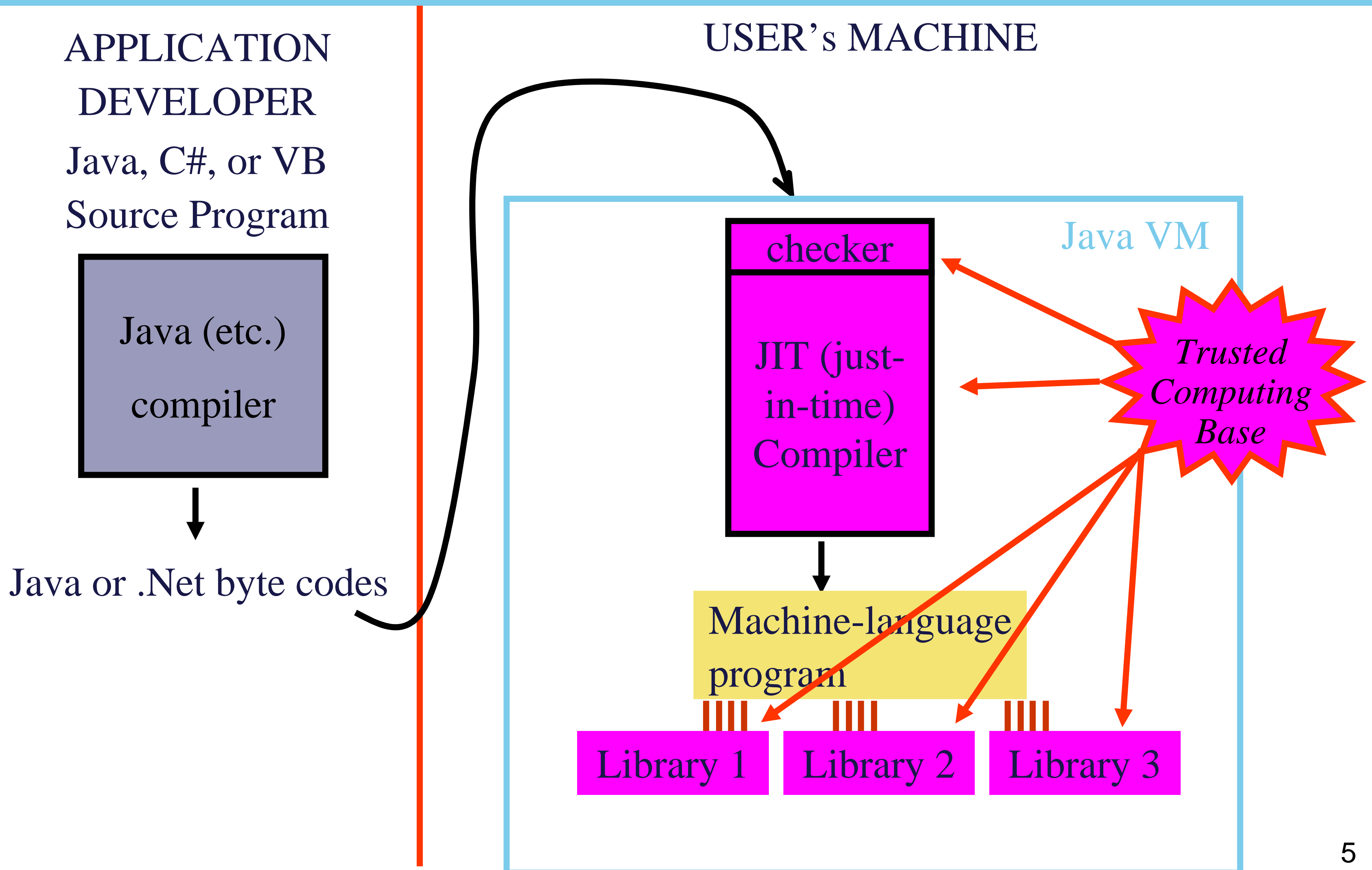
☺ Fine-grain access-control possible

- ☺ Very expressive policy languages possible

☺ Specified in terms the programmer understands

☹ Complicated to implement securely

Java or .Net virtual machine

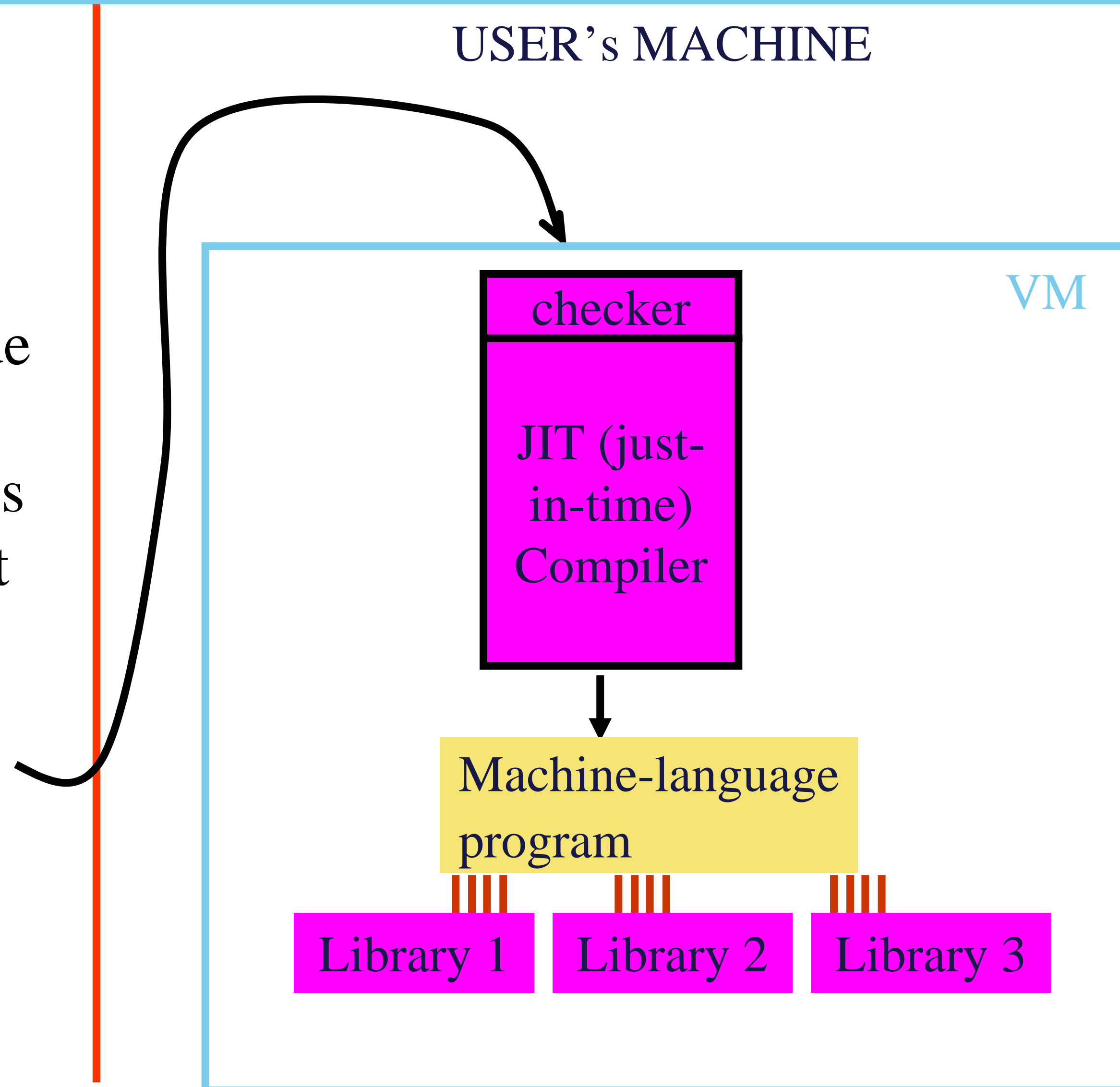


Outline of talk

- Securing the JIT compiler
- Securing the libraries
- Securing the application

1. How to attack the JIT!

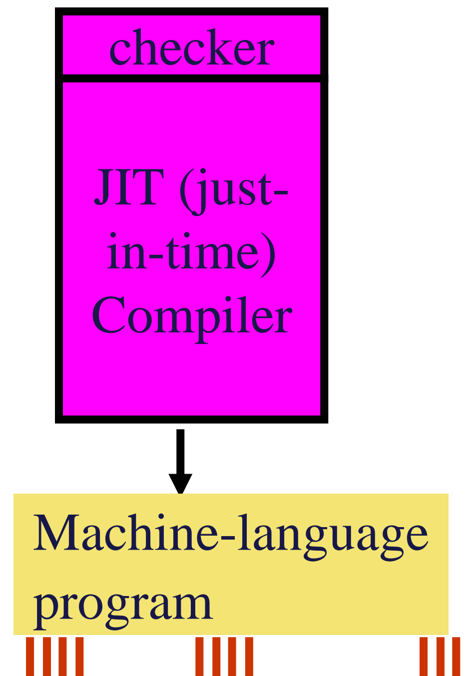
1. Find a bug in the JIT (every user has a copy!)
2. Write byte-code that tickles the bug (i.e., passes the checker but generates machine code that “cheats”)



Solution: prove a soundness theorem!

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

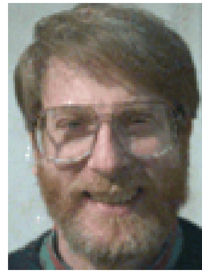
Proof: ?



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

People kept at it, though...

Type Systems Research

1978-2004

Provably* sound type systems; "Well-typed programs can't go wrong."

*in the mathematician's sense

ML, Modula-3, Java...

(each one an advance over its successors!)

Mechanical Proof

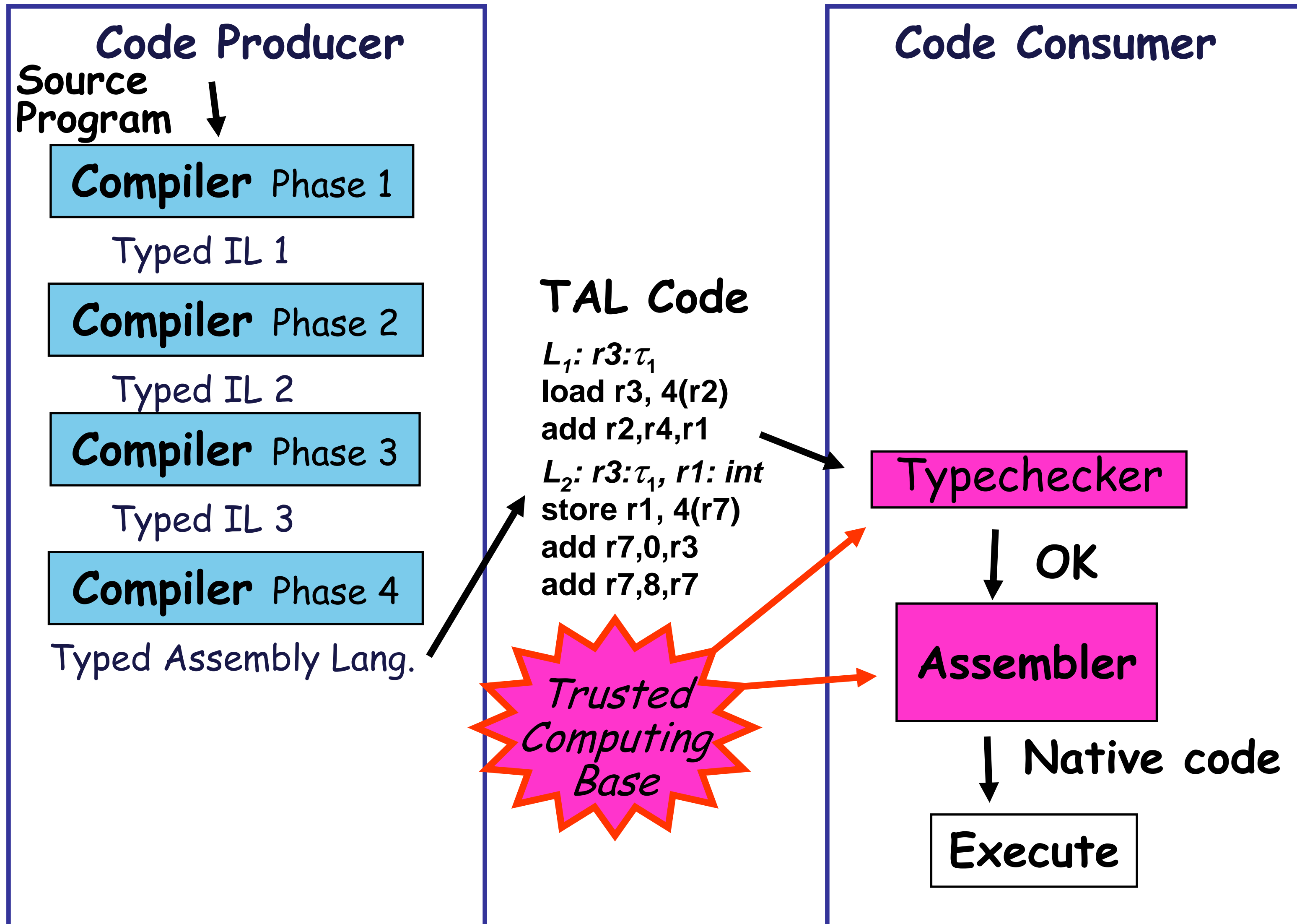
1978-2004

(not "program verification") but proof-development and proof-checking tools

Boyer-Moore, HOL, Isabelle, NuPrl, Coq, ...

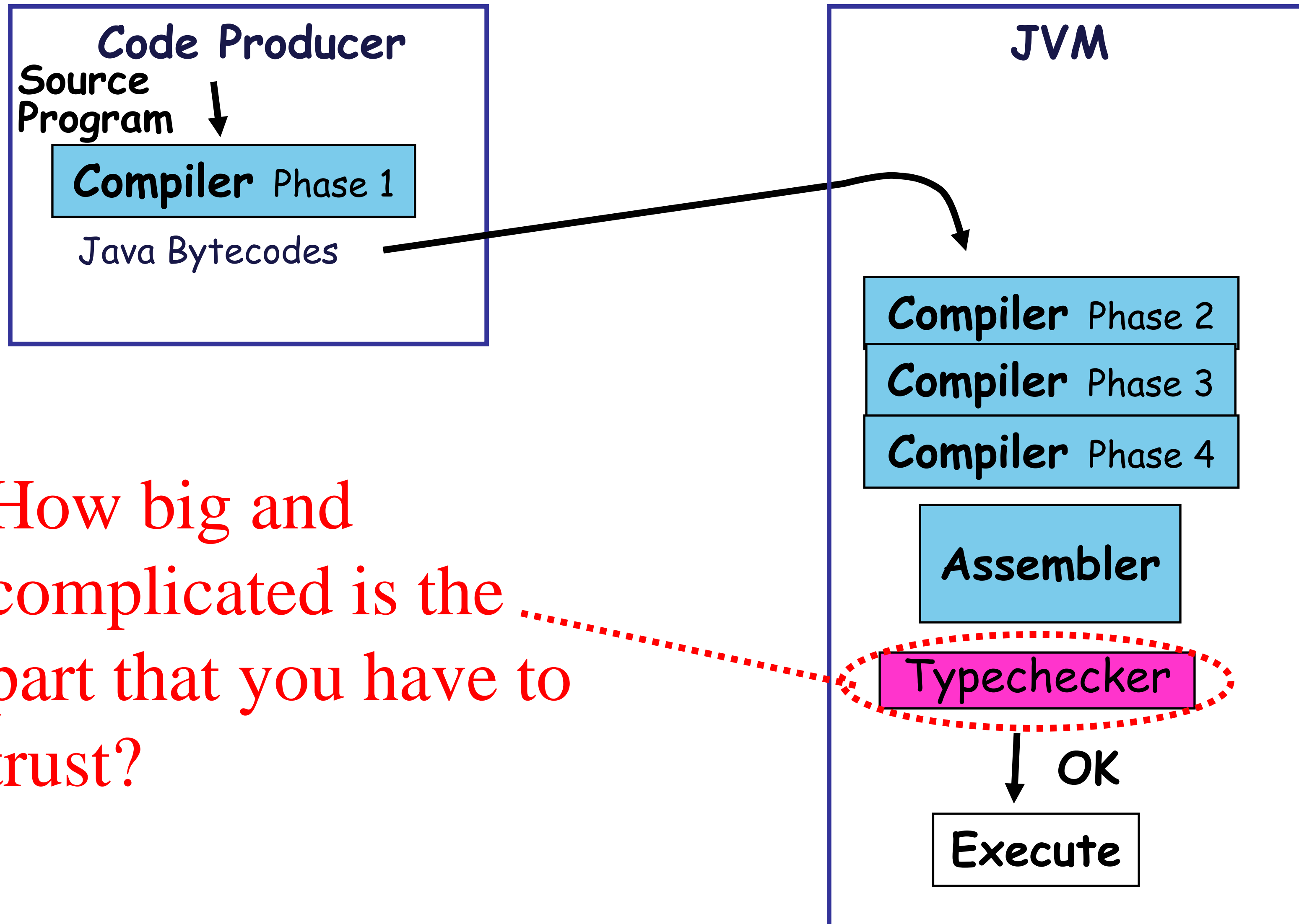
(Necula '97; Morrisett '98)

Typed Assembly Languages / PCC



Java or .Net compatible

Colby et al., PLDI '00



Size of Trusted Computing Base

Princeton FPCC compiler *(to scale!)*

Pre-existing components

Custom components

Standard ML of New Jersey front end

Type-preserving middle

MLRISC retargetable code generator

170k lines of code

TAL

Sparc code

Checker

Execute

Juan Chen

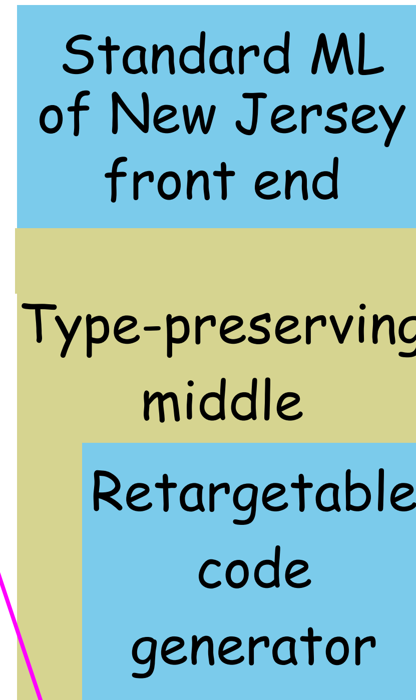
A Low-Level Typed Assembly Language with a Machine-checkable Soundness Proof

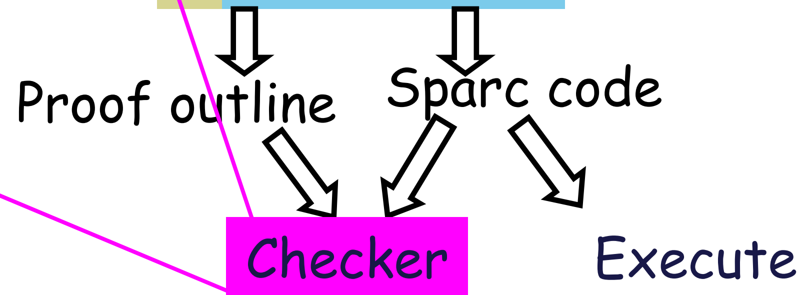
PhD Thesis, 2004

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





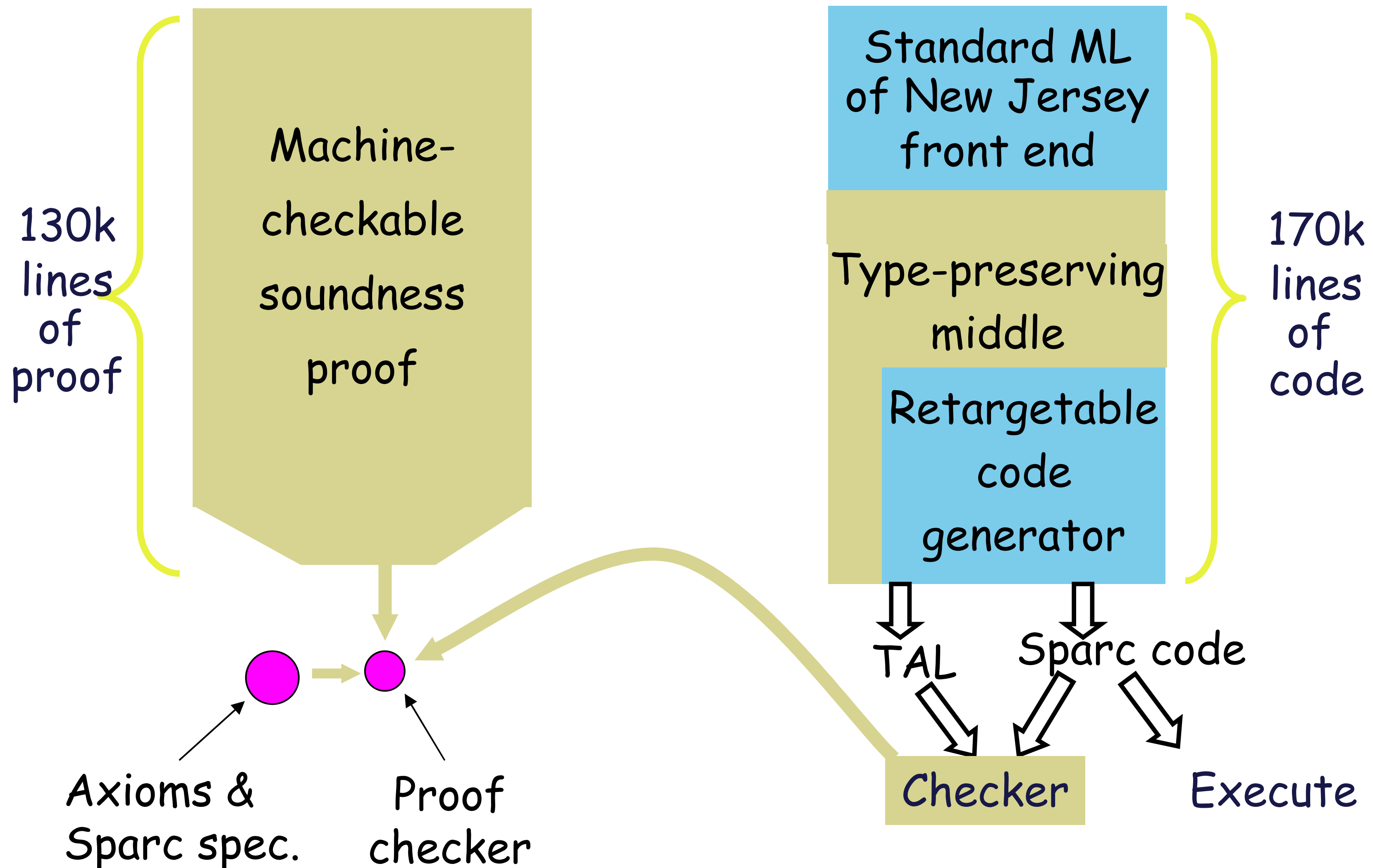
Here's one rule *(can you trust this?)*

(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

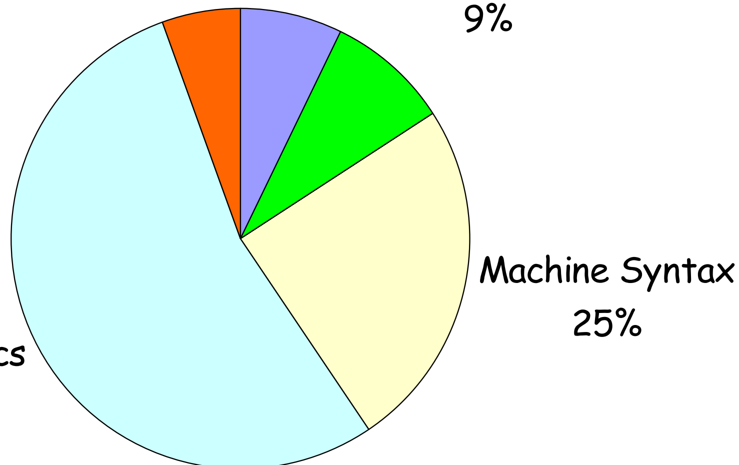
$$\frac{\begin{array}{l}\text{regbind } \Gamma \text{ At A} \\ \text{realreg At Ar} \\ \text{regbind_val } \Gamma \text{ B Bt} \\ \text{realreg Bt Br} \\ \text{match_reg_or_imm } \Gamma \text{ C Cx} \\ \text{valueTy } \Gamma\ \Phi \text{ N B int32} \\ \text{valueTy } \Gamma\ \Phi \text{ N C int32} \\ \text{venv_add}\backslash\ \Gamma \text{ A int32 } \Phi\ \Phi' \\ \text{decode_list L L' P P' (i_ADD Br Cx Ar)}\end{array}}{\begin{array}{l}\Gamma \Rightarrow \text{L M Q N P } \Phi \\ \qquad \text{(e_prim A (p_arith a_add B C))} \\ \qquad \text{L' M Q N P' } \Phi'\end{array}}$$

Machine-checkable proof *(to scale!)*

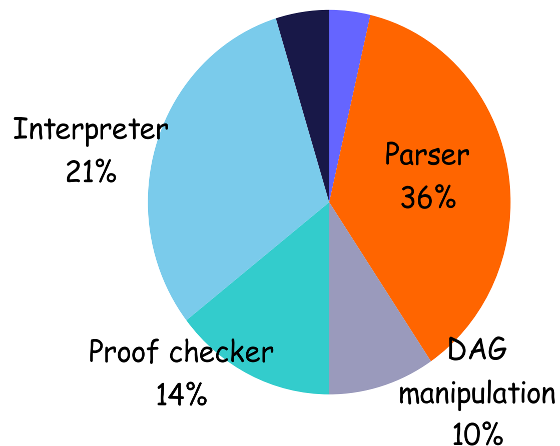


Size of Trusted Code Base: 3028 lines

Policy & Specification



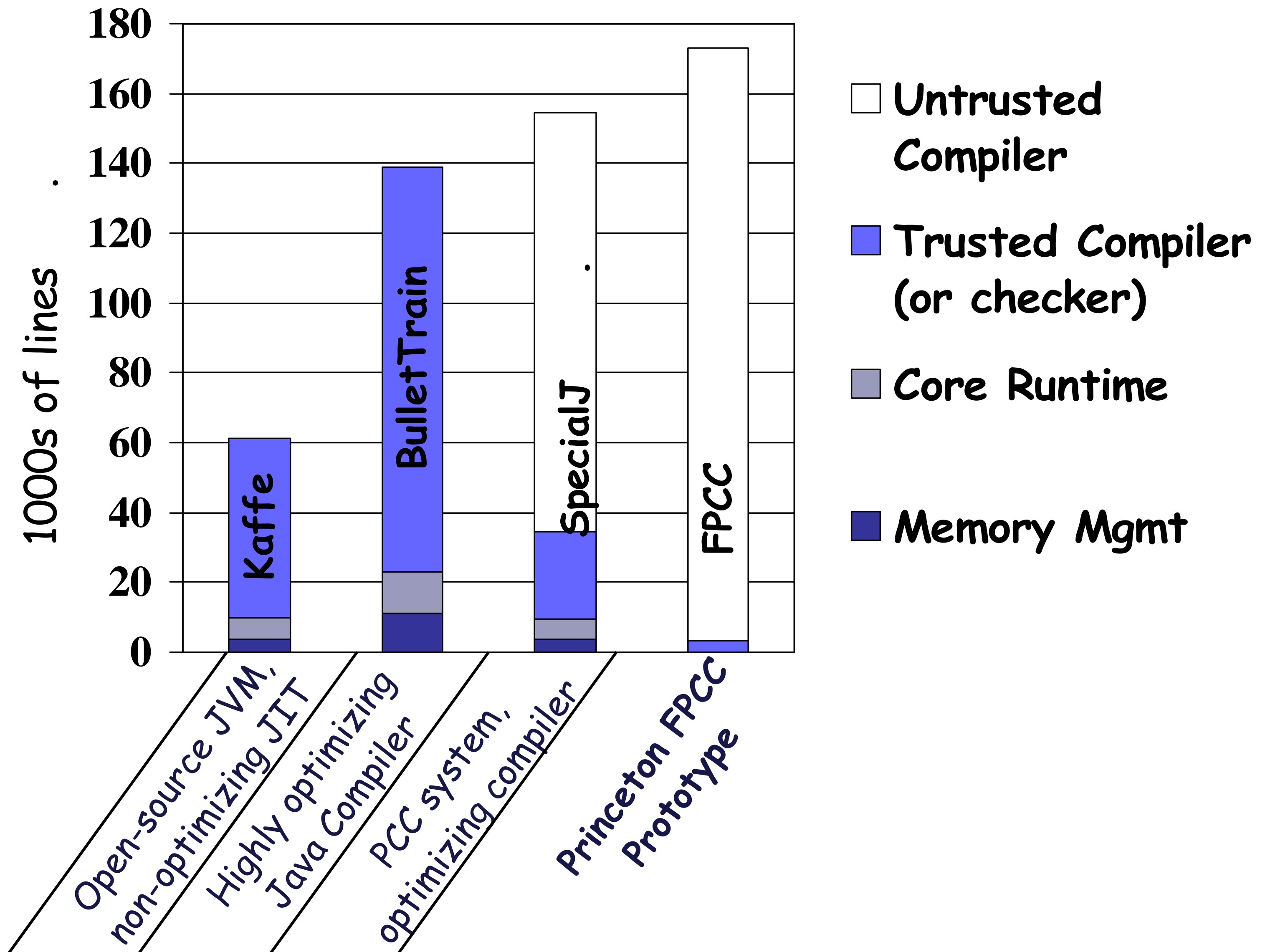
General-purpose Proof checker ("Flit")



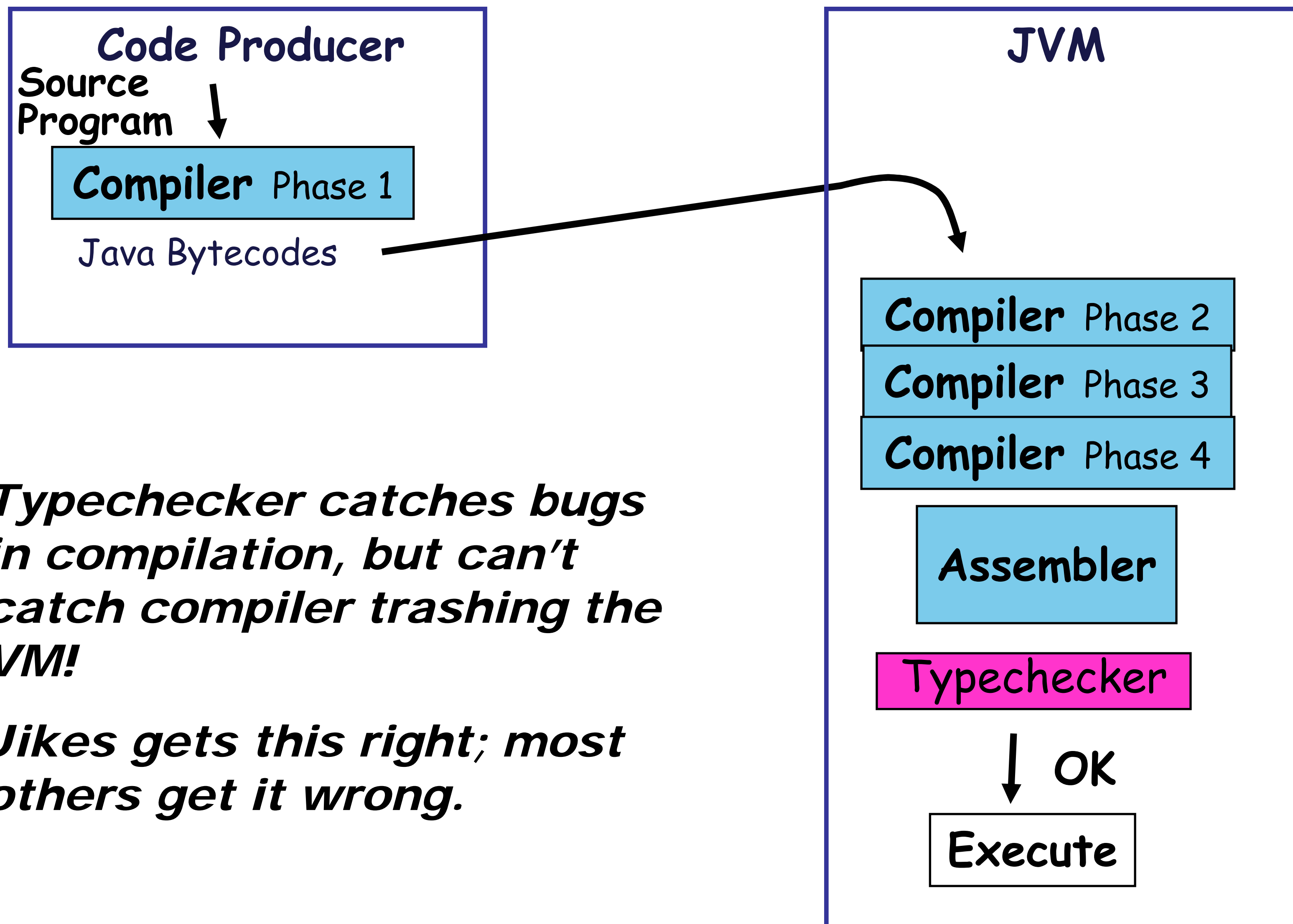
Component	Lines
Logic	135
Arithmetic	160
Machine Syntax	460
Machine Semantics	1005
Safety Predicate	105
Total	**1865**

Component	Lines
Input/Output	43
Parser	428
DAG creation	111
Type checking	167
Interpreter	360
Main program	54
Total	1163

Smallest possible Trusted Base



Write JIT in type-safe VM language!

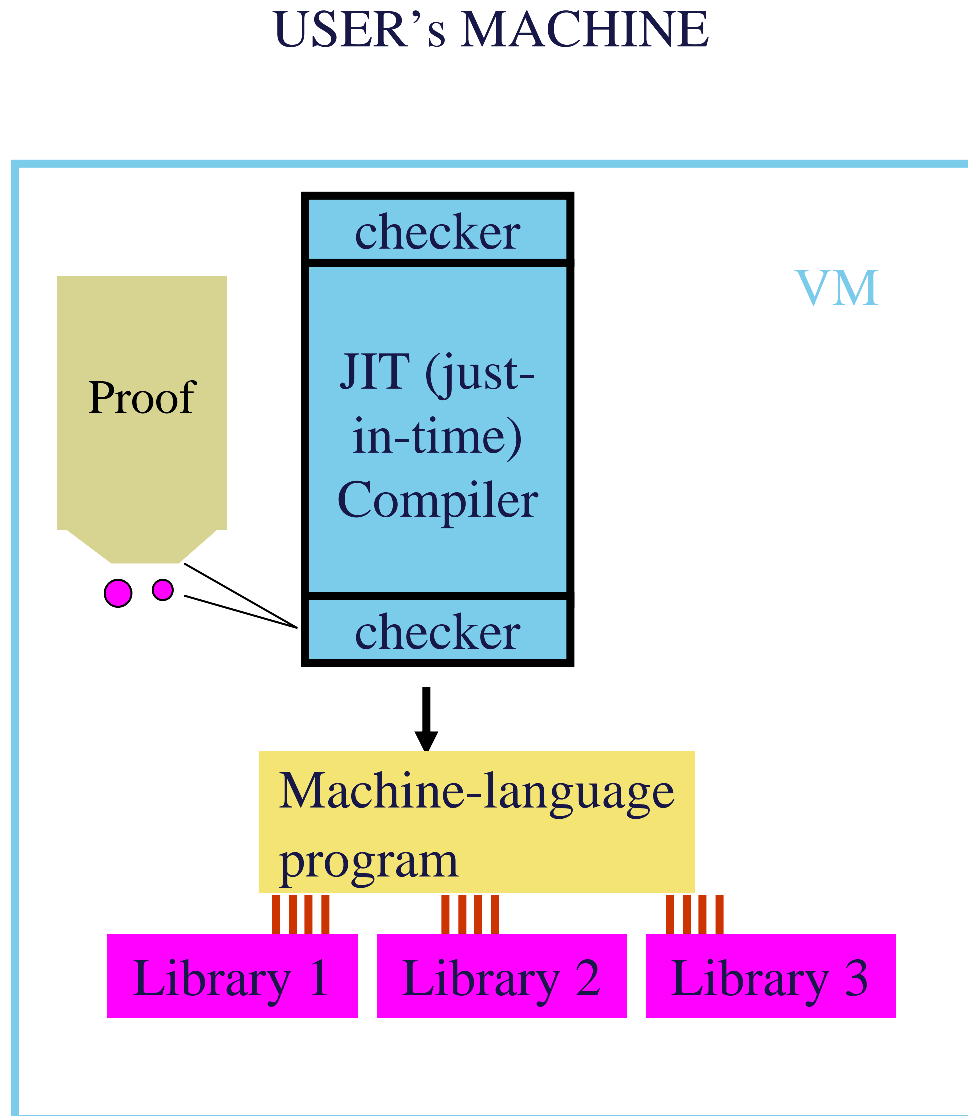


Typechecker catches bugs in compilation, but can't catch compiler trashing the VM!

Jikes gets this right; most others get it wrong.

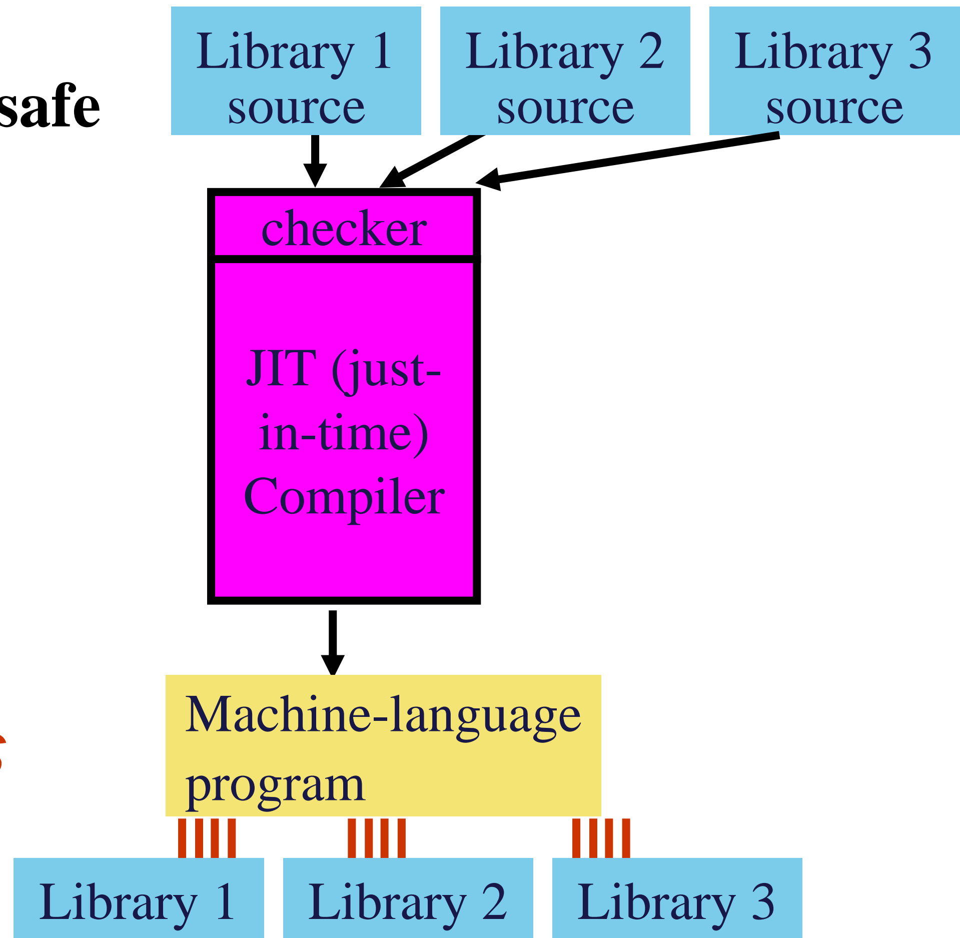
2. How to attack the libraries!

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



Solution (?)

Write libraries in the type-safe source language.

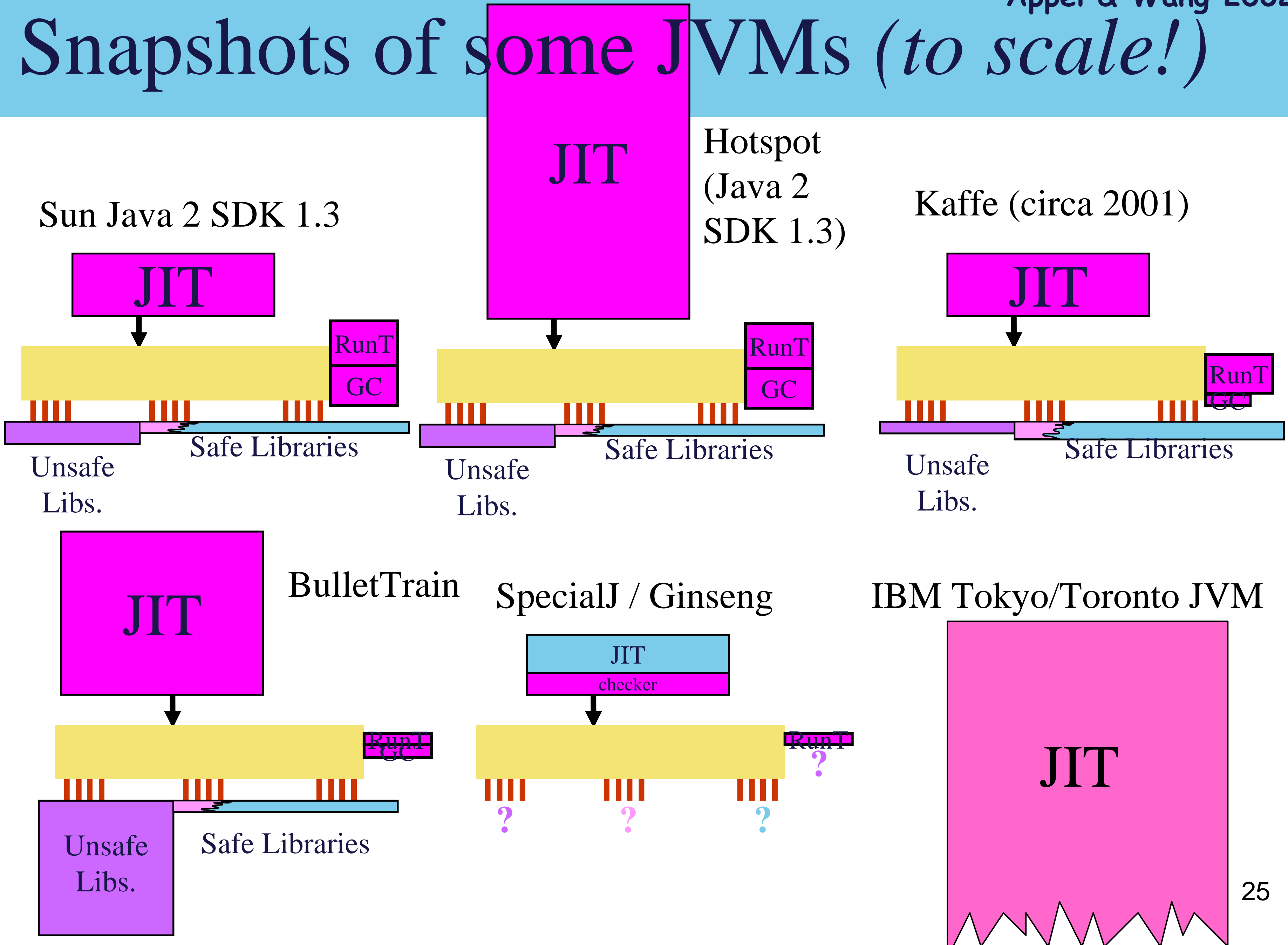


Not always possible!

Doesn't always guarantee security!

Three categories of library code

TYPE-UNSAFE	TYPE-SAFE SECURITY-RELEVANT	TYPE-SAFE USER LIBRARIES
Native-code libraries to support APIs not implementable in type-safe source language (Example: JNI)	Implemented in type-safe source language, provide sensitive capabilities such as I/O or security primitives	Not called by security-relevant libraries; utilities such as quicksort or GUI



Appel & Wang 2002
Snapshots of some JVMs (to scale!)
Sun Java 2 SDK 1.3
JIT
RunT
GC
Unsafe Libs.
Safe Libraries
Hotspot (Java 2 SDK 1.3)
JIT
RunT
GC
Unsafe Libs.
Safe Libraries
Kaffe (circa 2001)
JIT
RunT
GC
Unsafe Libs.
Safe Libraries
BulletTrain
JIT
RunT
GC
Unsafe Libs.
Safe Libraries
SpecialJ / Ginseng
JIT
checker
RunT
?
?
?
?
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

Securing JAVA

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!

Useful ideas, good in any context

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
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

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

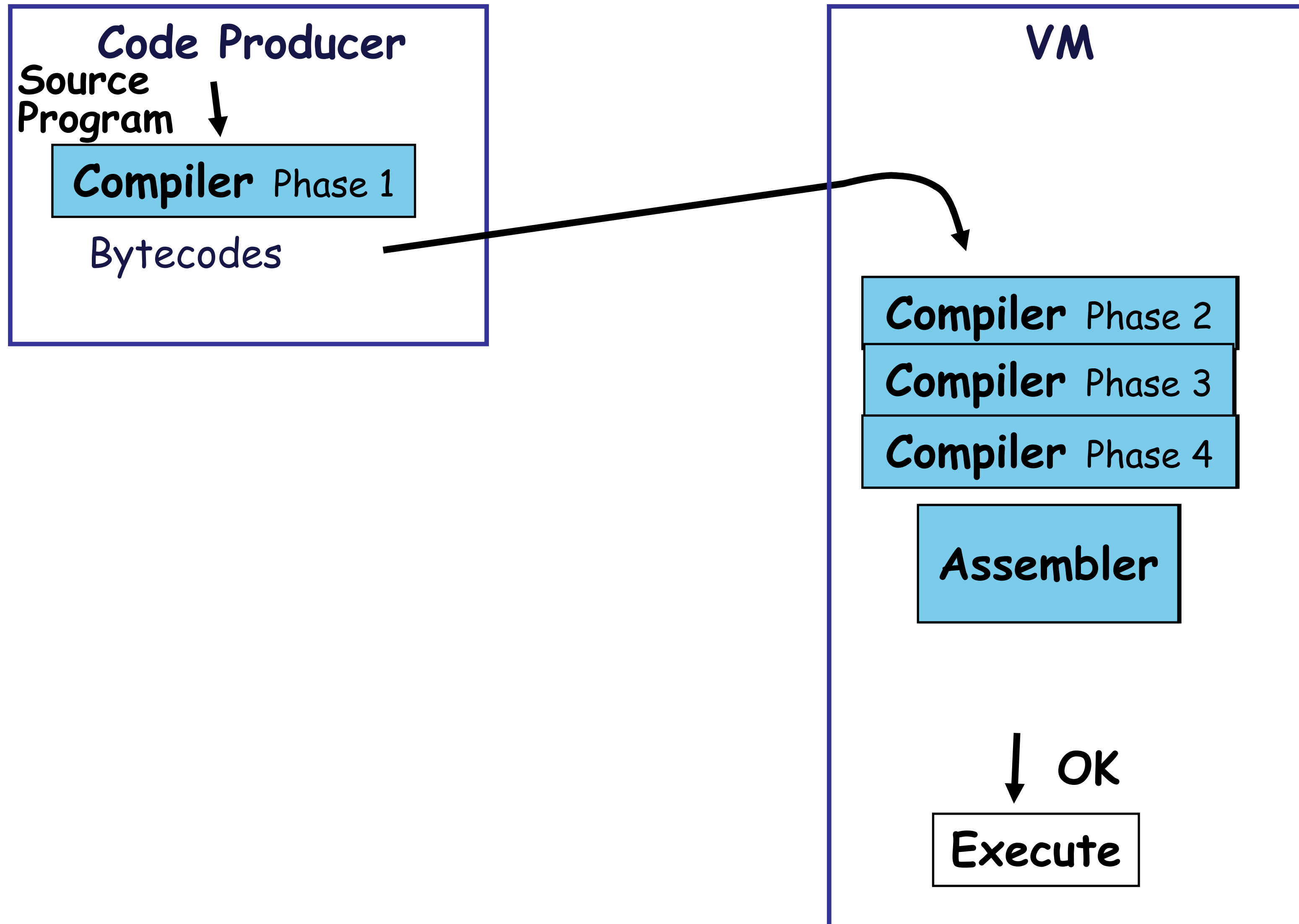
  (so executing the compiler won't be exploitable)

- 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



Advertisement

- 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.