High-Assurance
Java Virtual Machine

Alessandro Coglio

Kestrel Institute

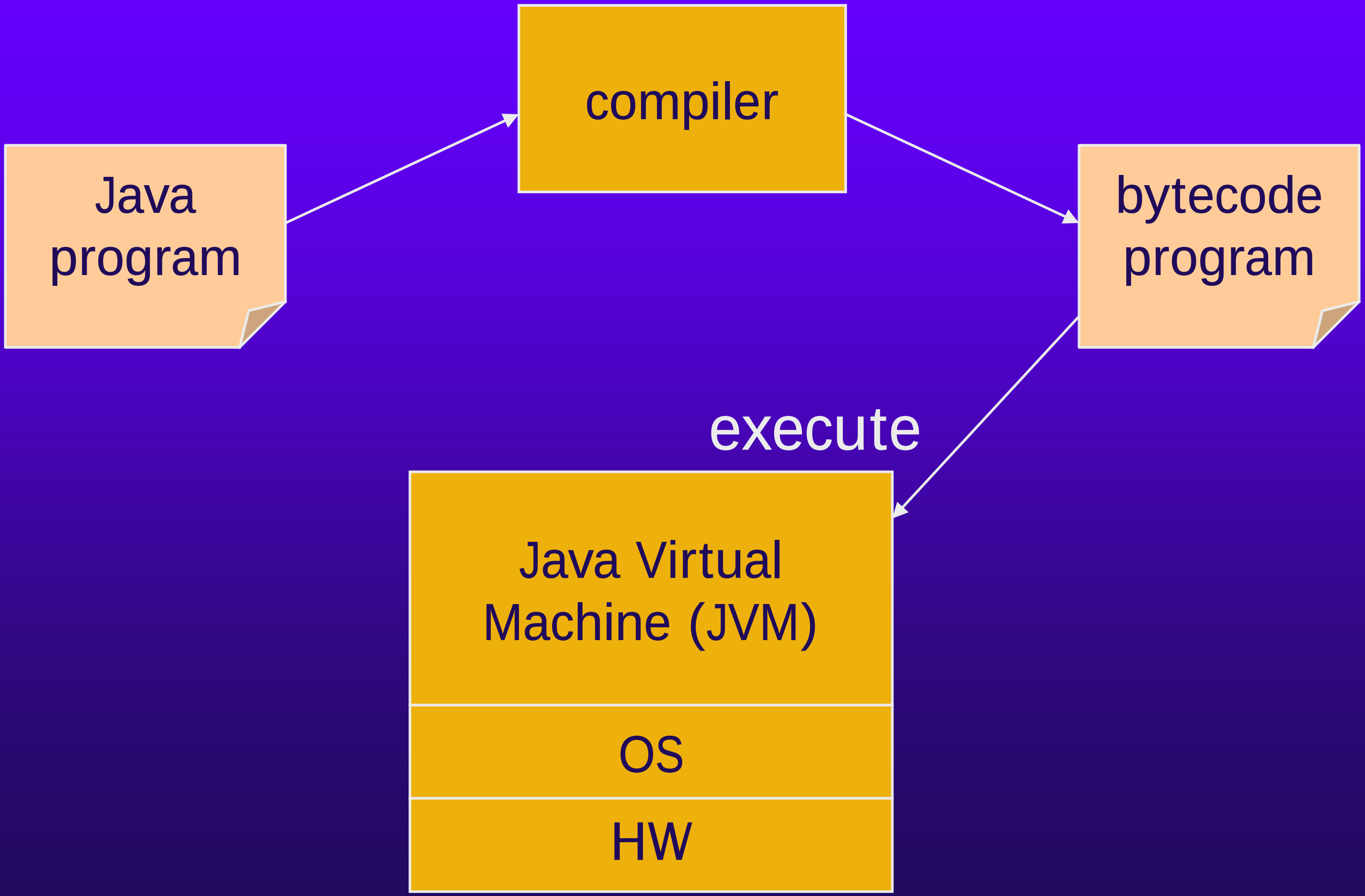
Summary

- Introduction and goals
- Type safety and security
- Bytecode verification
- Class loading
- Java Card

Summary

- Introduction and goals
- Type safety and security
- Bytecode verification
- Class loading
- Java Card

Java Virtual Machine (JVM)



Java program

compiler

bytecode program

execute

Java Virtual Machine (JVM)

OS

HW

JVM Security Mechanisms

- Protect HW/OS resources
  files, memory, devices, ...
- Support secure applications
  authentication, encryption, access control, ...
- Guarantee integrity of the JVM
  JVM always works as expected

Achieving High Assurance

- Assess that JVM design has the intended properties
  - precise description (spec) of the JVM
  - analysis of the description to
    - assess properties
    - discover flaws and find fixes
- Implement JVM correctly

  implementation code verifiably correct w.r.t. description above

Current Situation

- T. Lindholm, F. Yelling, “The Java Virtual Machine Specification” (2nd ed.) from Sun
  - informal English prose
  - well written but contains ambiguities
    ⇒ hard to assess properties
- Sun’s Java 2 SDK vers. 1.3
  - reference implementation in C
  - precise but not very readable
    ⇒ hard to assess properties
  - no verifiable “connection” with spec above

No high assurance!

Specware

- Precise, formal specs
- Automated refinements to code
- Composition of specs and refinements
- Mechanical proofs of
  - desired properties
  - correctness of refinements

JVM in Specware

- Precise, formal specs
- Automated refinements to code
- Composition of specs and refinements
- Mechanical proofs of
  - desired properties
  - correctness of refinements

JVM in Specware

- Precise, formal specs
  Specify the JVM
- Automated refinements to code

- Composition of specs and refinements
- Mechanical proofs of
  - desired properties
  - correctness of refinements

JVM in Specware

- Precise, formal specs
  Specify the JVM
- Automated refinements to code
  Derive provably-correct implementation
- Composition of specs and refinements
- Mechanical proofs of
  - desired properties
  - correctness of refinements

Benefits

- Description of the JVM that is
  - precise (formal)
  - readable
    - structured, compositional
    - multiple levels of abstraction
  - easier to assess properties about
- Implementation of the JVM that is provably correct w.r.t. description above

High assurance!

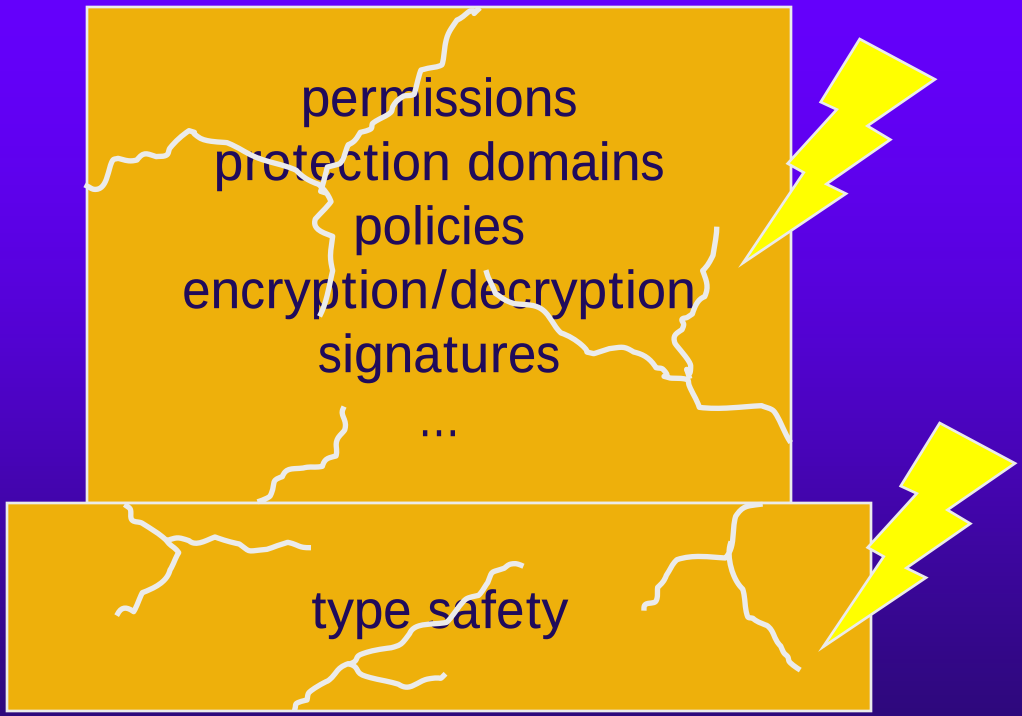
Summary

- Introduction and goals
- Type safety and security
- Bytecode verification
- Class loading
- Java Card

JVM Security...

permissions
protection domains
policies
encryption/decryption
signatures
...

… Is Built Upon Type Safety



permissions
protection domains
policies
encryption/decryption
signatures
…

type safety

if type safety is broken...

...security is also broken

What is Type Safety?

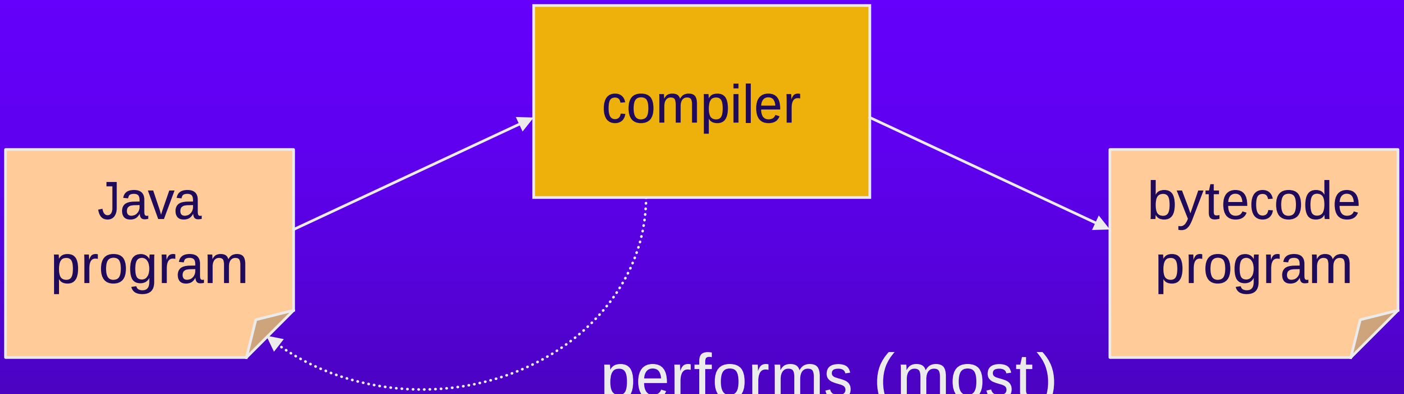
Data are always manipulated consistently with their type, e.g.

- method call **`o.m()`** requires
  - **`m`** to be declared in or inherited by class of **`o`**
  - caller to have access to **`m`**
- memory cannot be randomly accessed through pointers (unlike C/C++)
- array index **`i`** in **`a[i]`** must be within bounds (i.e., **`0 <= i < a.length`**)

How Is Security Based on It?

- JVM security mechanisms assume type safety, e.g.
  - no buffer overflows (~50% of attacks)
  - **`private byte[] secret_key`** cannot be accessed outside its class (unlike C++)
  - HW/OS only accessible through “controlled” fields/methods, not directly
- Flawed design or implementation of type safety mechanisms allows security mechanisms to be bypassed

Enforcing Type Safety



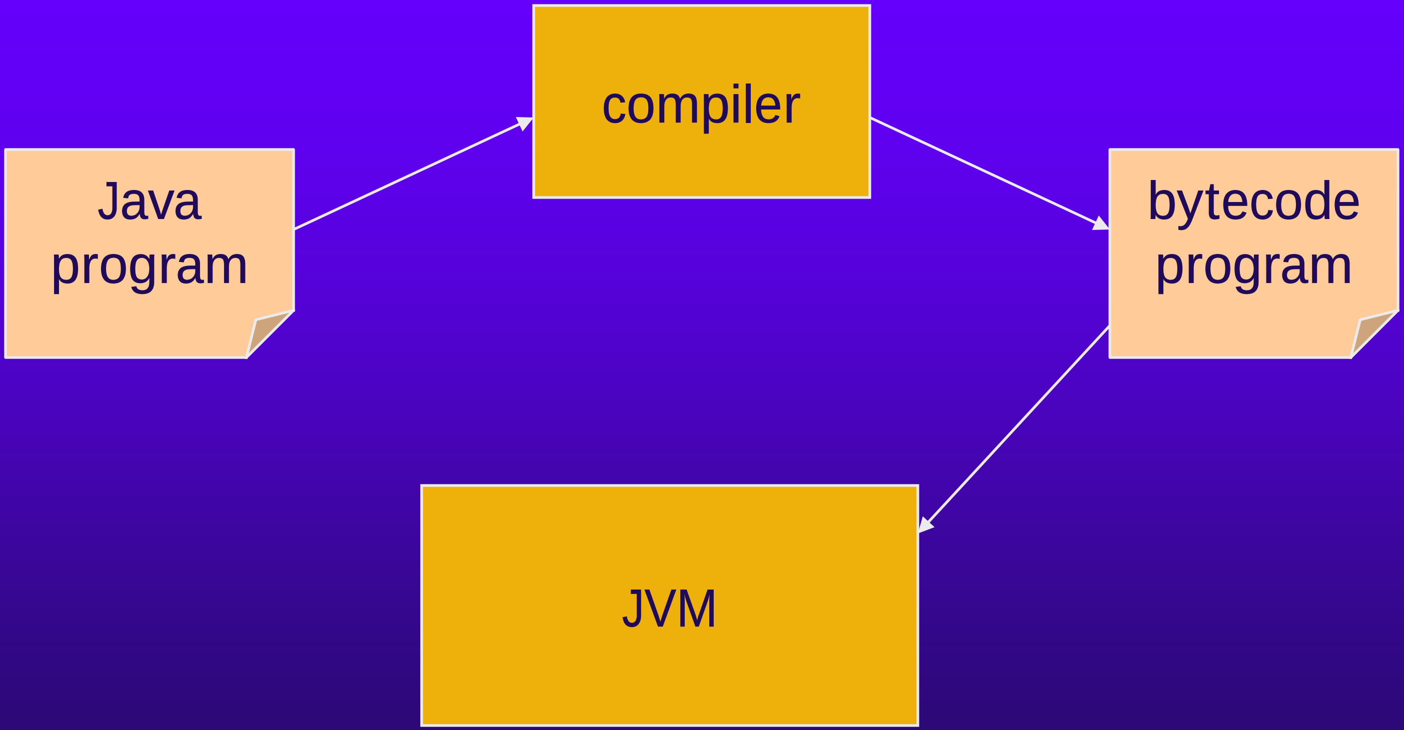
Java program

compiler

bytecode program

performs (most) type safety checks

Enforcing Type Safety



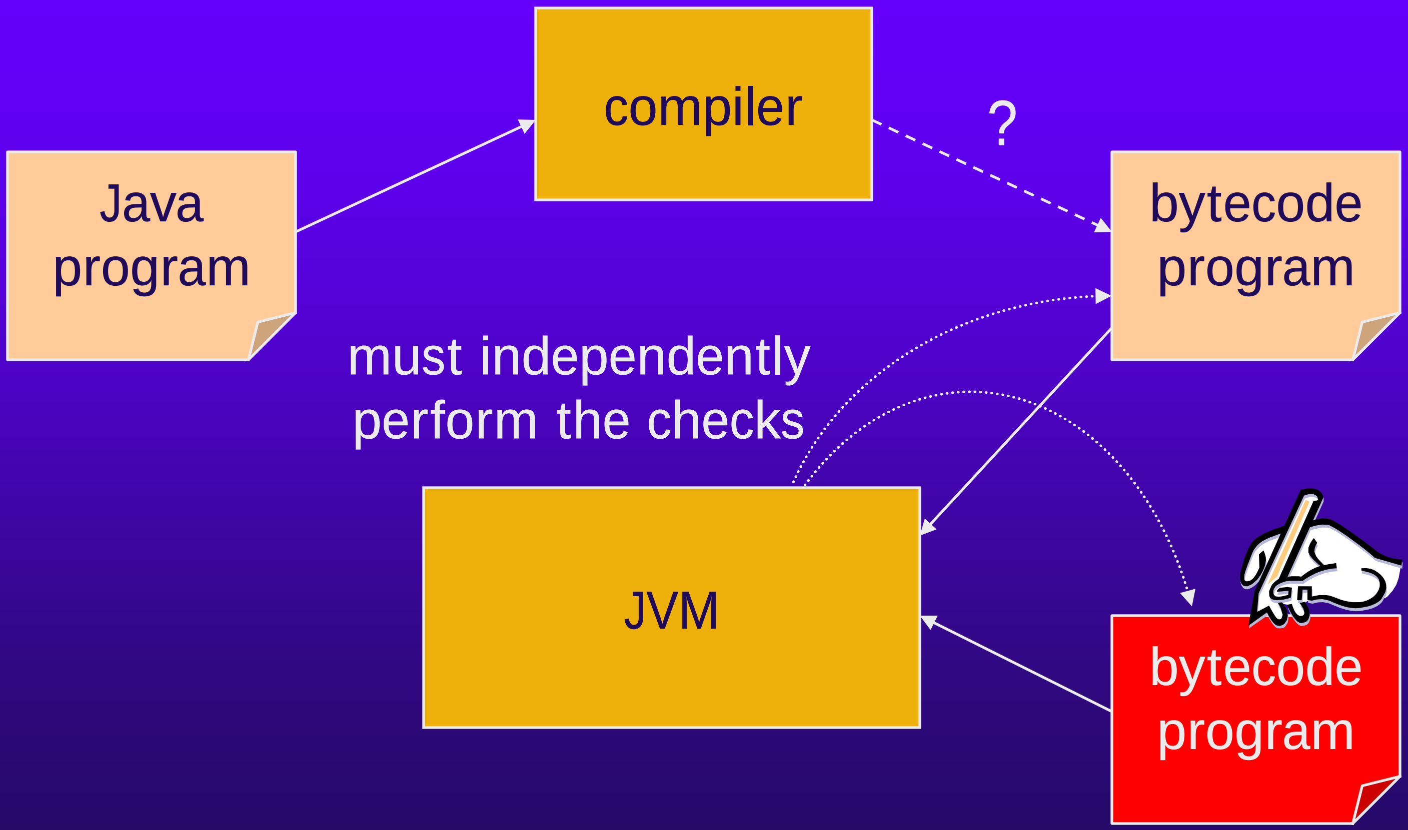
compiler

Java program

bytecode program

JVM

Enforcing Type Safety



compiler

Java program

?

bytecode program

must independently perform the checks

JVM

bytecode program

from the Internet

Where Is the Problem?

- There are very tricky points
- Bugs have been found in the JVM (design & implementation) that violate type safety
  - Saraswat, 1997
  - Tozawa & Hagiya, 1999
  - Coglio & Goldberg, 2000
- Current release (SDK 1.3) is not type-safe

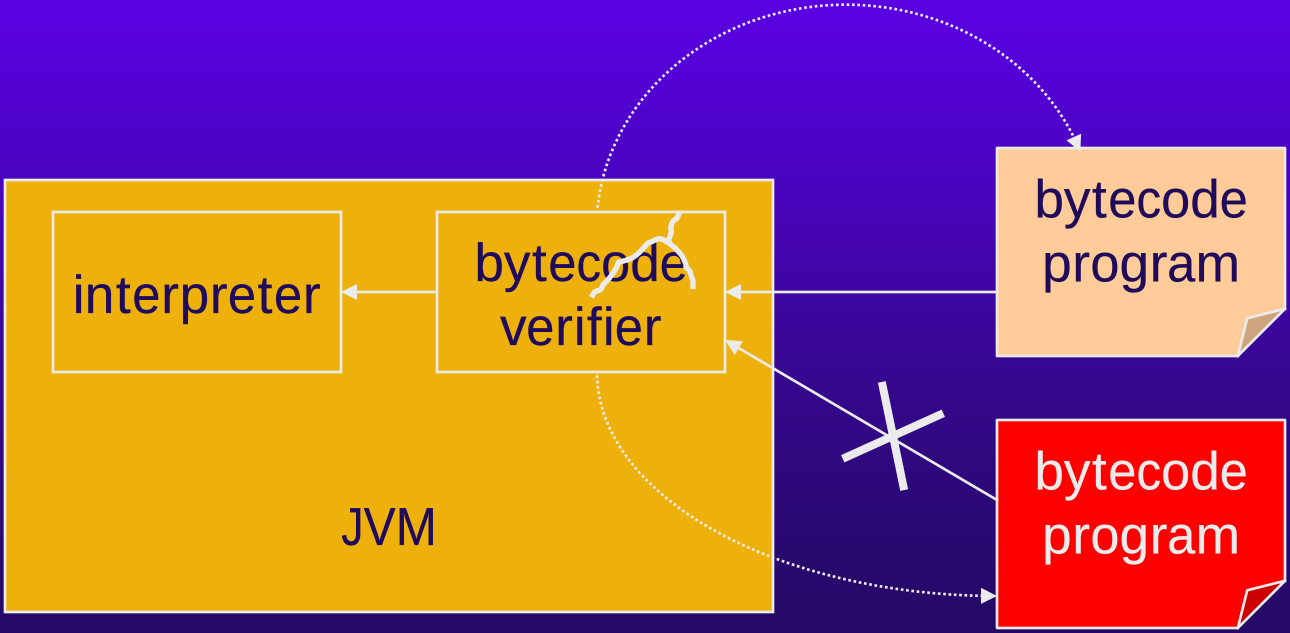
  but applets cannot directly exploit these known bugs

Summary

- Introduction and goals
- Type safety and security
- Bytecode verification
- Class loading
- Java Card

Bytecode Verification

BV is a critical component

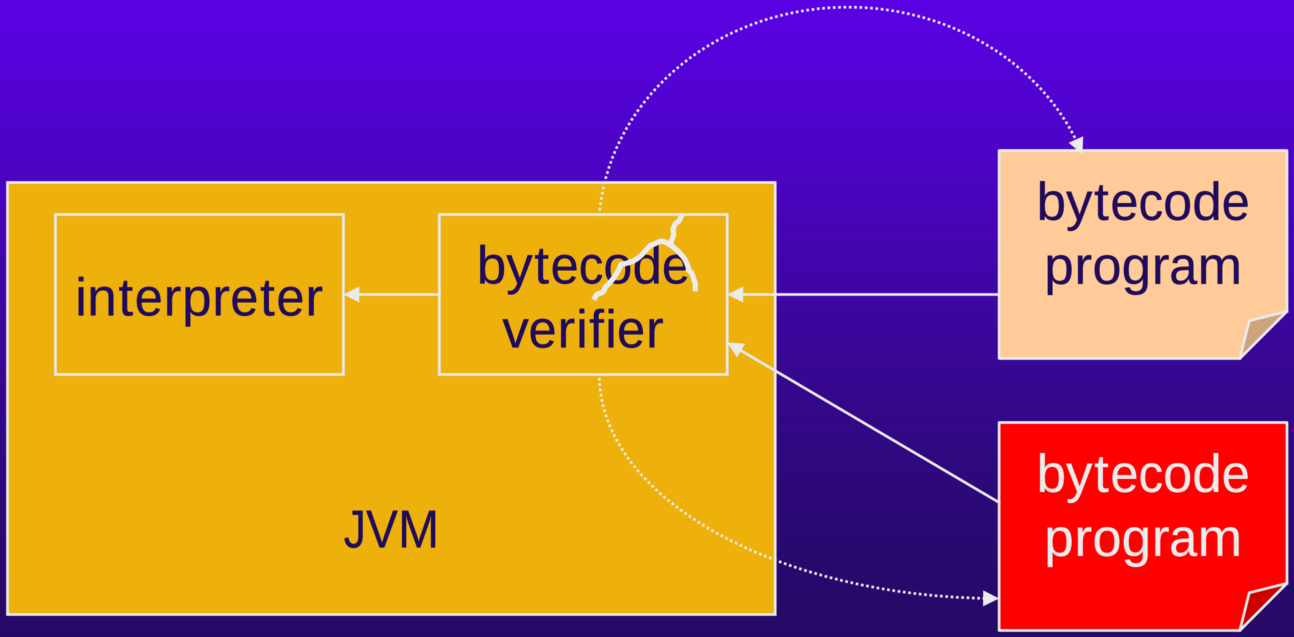


static type safety checks
(~ equivalent to compiler's)

Bytecode Verification

BV is a critical component

static type safety checks
(~ equivalent to compiler's)



interpreter

bytecode verifier

JVM

bytecode program

bytecode program

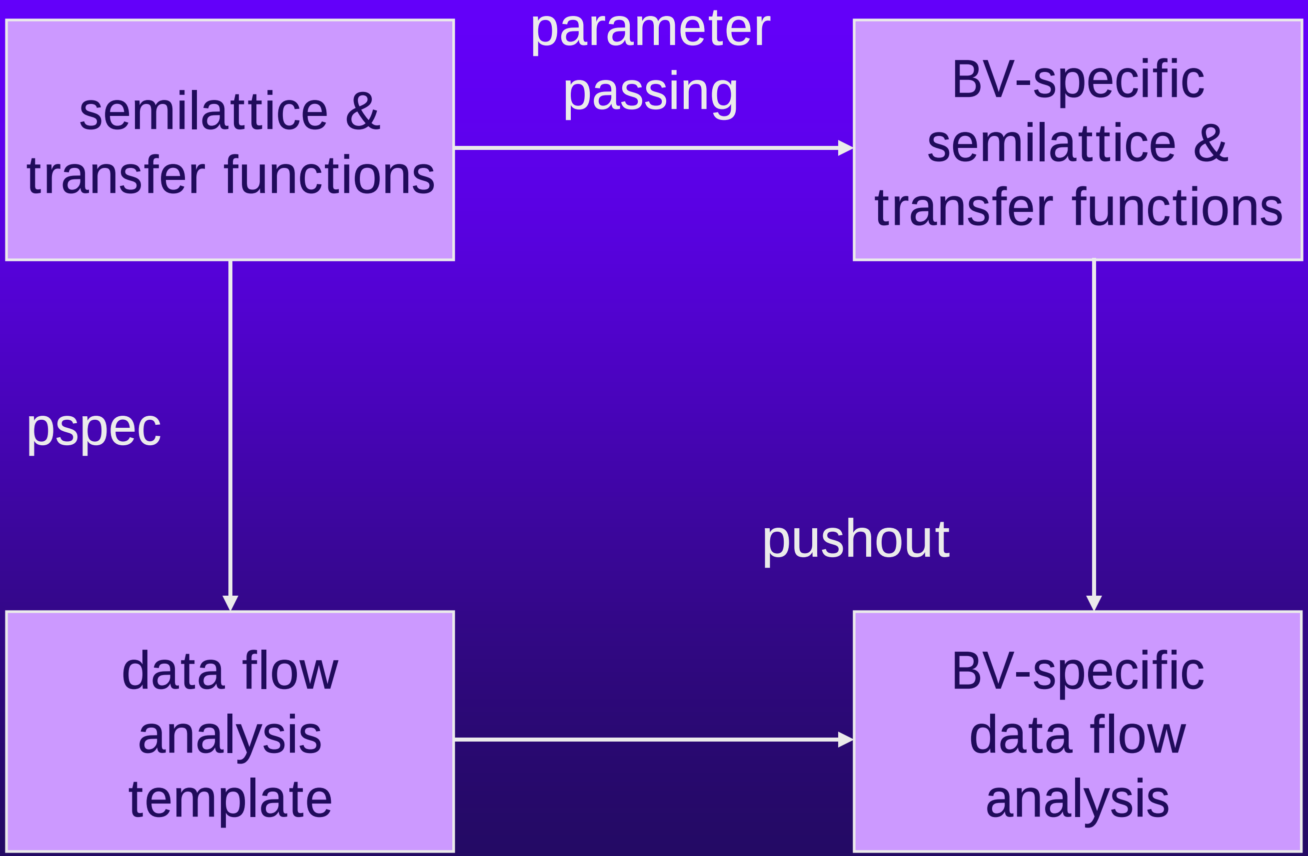
Our Achievement: Complete BV in Specware

- Precise, readable, compositional spec
  - ~3K loc in Metaslang (vs. Sun's ~4K loc in C)
- Refinement to running code
- Successfully tested on 4K+ classes
- Amenable to optimization (via Specware refinements)
- Manual type safety proofs
- See demo tomorrow!

Highlights of Our BV

- Type inference via data flow analysis
  - type t assigned to each local memory location x at each bytecode instruction i
  - at run time, value in x at instruction i is t
  - no consistent assignment ⇒ program rejected
- Instantiation of data flow analysis template
  - data flow structure explicit
  - in Sun’s BV data flow structure is buried in code

Instantiation of Data Flow Analysis Template



semilattice & transfer functions

parameter passing

BV-specific semilattice & transfer functions

pspec

pushout

data flow analysis template

BV-specific data flow analysis

Leverage of Our BV

- Re-usable components
  - data flow analysis engine
  - types
  - ...
- Basis for other bytecode analyses
  - vulnerability detection
  - defect finding
  - ...

Main Difficulty in BV: Subroutines

- Two bytecode instructions
  - **`jsr`** jump to subroutine
  - **`ret`** return from subroutine
- Used by compilers to reduce code replication in bytecode

  not visible at the Java source level

Problem with Subroutines

- For accurate type inference, flow of control of subroutines must be properly taken into account
- However, subroutines
  - ... may not be textually delimited
  - ... may not have LIFO behavior
  - ... may be exited not through a **`ret`**
    - branch
    - exception

Sun's Treatment of Subroutines

- Spec
  - complicated
  - not completely defined
  - includes unnecessary restrictions
- Implementation
  - does not fit data flow analysis framework $\Rightarrow$ harder to prove properties
  - contradicts spec above (e.g., recursion)
  - <u>rejects some compiled programs</u>

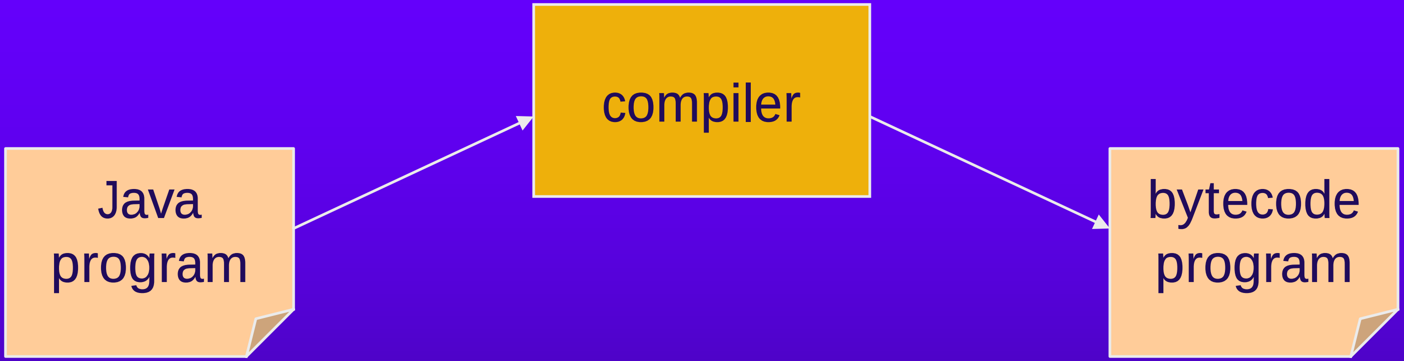
Our Treatment of Subroutines

- Two novel techniques to treat subroutines
  - much simpler and clearer
    $\Rightarrow$ higher assurance
  - accept all compiled programs
  - accept programs currently not produced by compilers (new possibilities for compilers)
  - #1 accepts more programs,
    #2 is more efficient and closer to Sun’s
- Useful contribution to other developers too

Summary

- Introduction and goals
- Type safety and security
- Bytecode verification
- Class loading
- Java Card

Class Loading

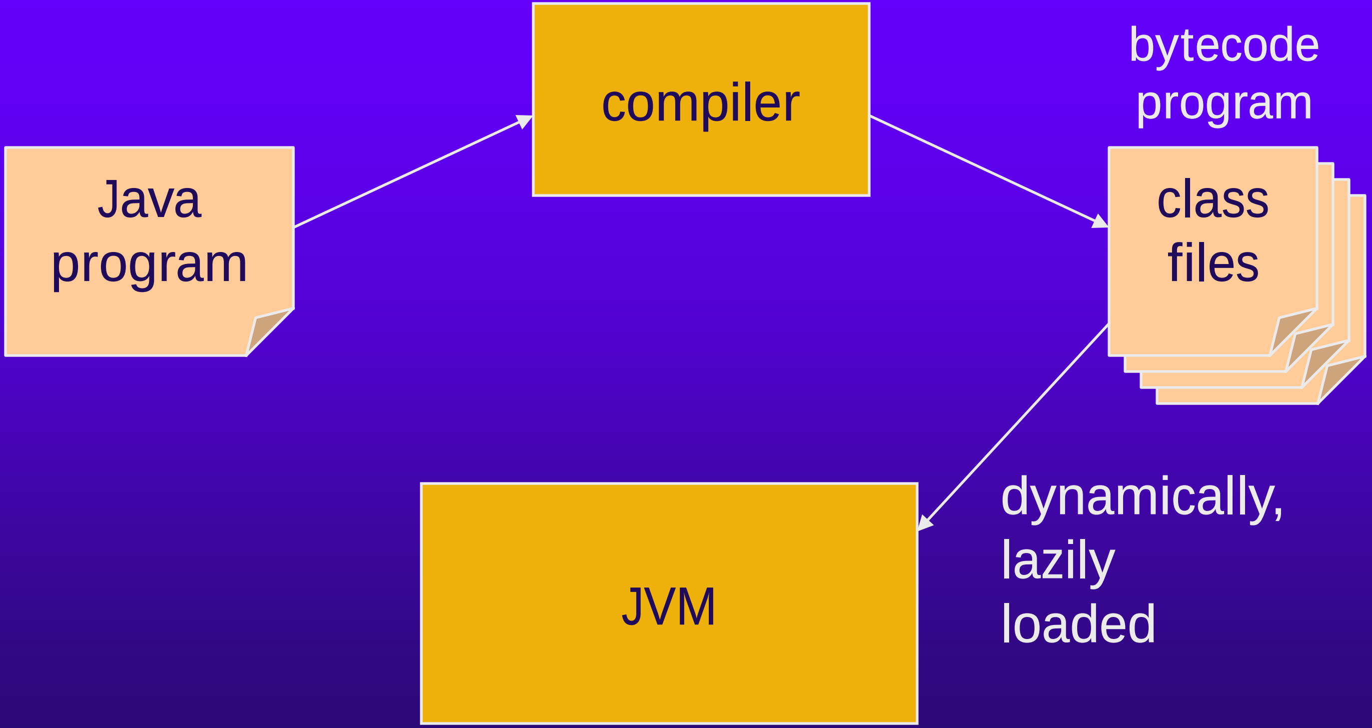


Java program

compiler

bytecode program

Class Loading



Java program

compiler

bytecode program

class files

dynamically, lazily loaded

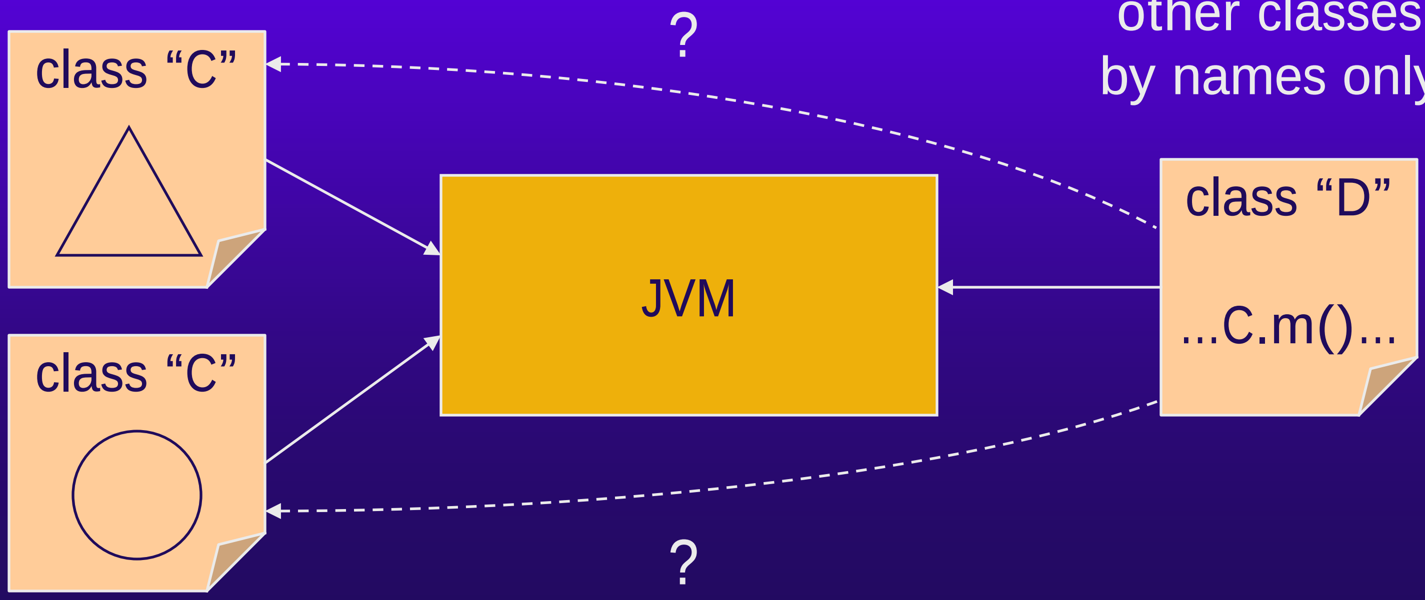
JVM

Multiple Name Spaces

different classes
with the same name
can be loaded into the
JVM (by different,
<u>user-defined</u> loaders)

potential
class spoofing
attacks!

classes reference
other classes
by names only



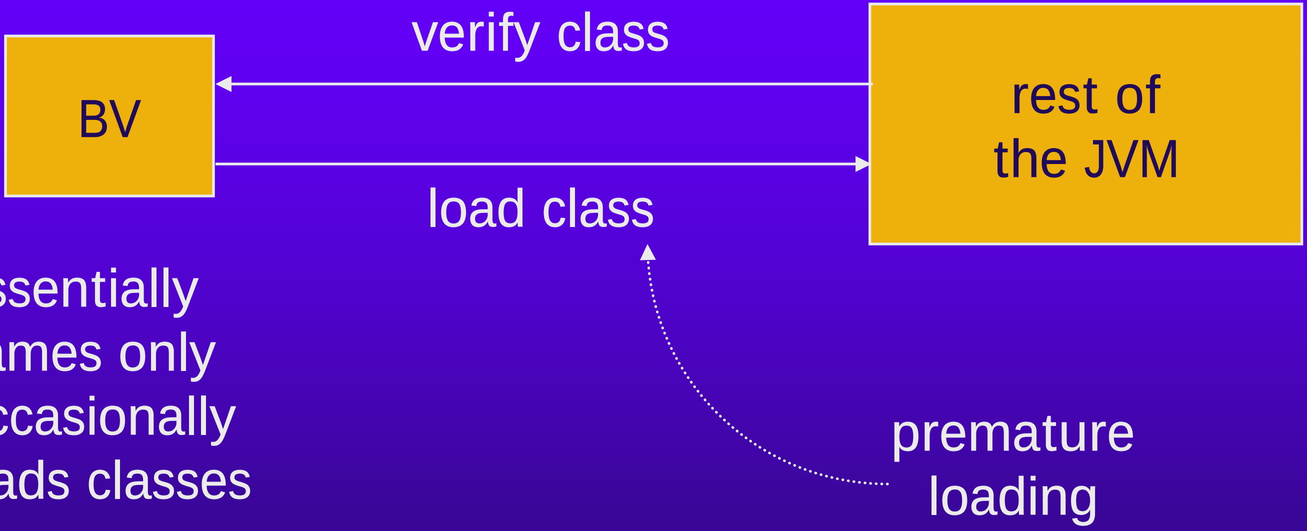
Enforcing Type Safety with Multiple Name Spaces

- BV is not enough because it deals with class names only, not classes
- Additional mechanisms are needed

Bugs Allowing Class Spoofing

- Saraswat, 1997
  - deficiencies in loading mechanisms
  - corrected by Sun’s introduction of loading constraints [Liang & Bracha, 1998]
- Tozawa & Hagiya, 1999
  Coglio & Goldberg, 2000
  - bugs in BV’s treatment of classes by names and in interaction between BV and loading
  - some “indirectly” corrected, others still there

Interaction BV ↔ Loading



BV

verify class

load class

rest of the JVM

premature loading

- essentially names only
- occasionally loads classes

Our Achievement: Better Interaction BV ↔ Loading

- BV uniformly uses names
  - precise disambiguation of names
  - does not load classes
- BV posts subtype constraints
  - lazily checked
  - integrated with Sun’s loading constraints
- BV is purely functional component of the JVM
- Employed by our BV in Specware

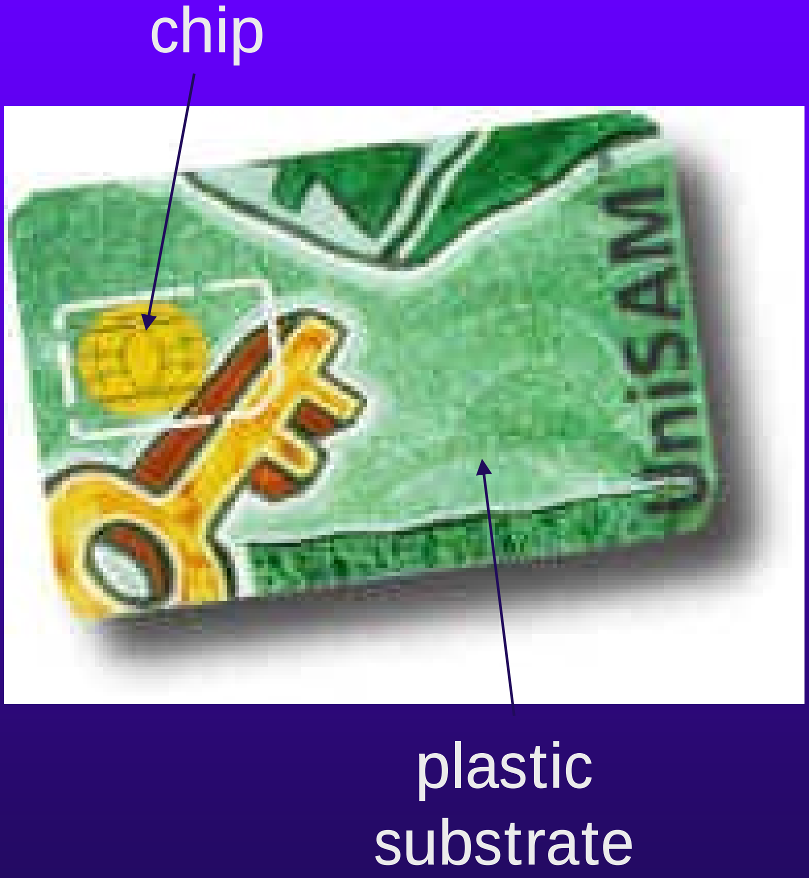
Our Achievement: Assessment of Type Safety Properties

- Mathematical formalization of
  - loading mechanisms
  - interaction of class loading with BV (according to our improved design)
- Type safety theorem

Summary

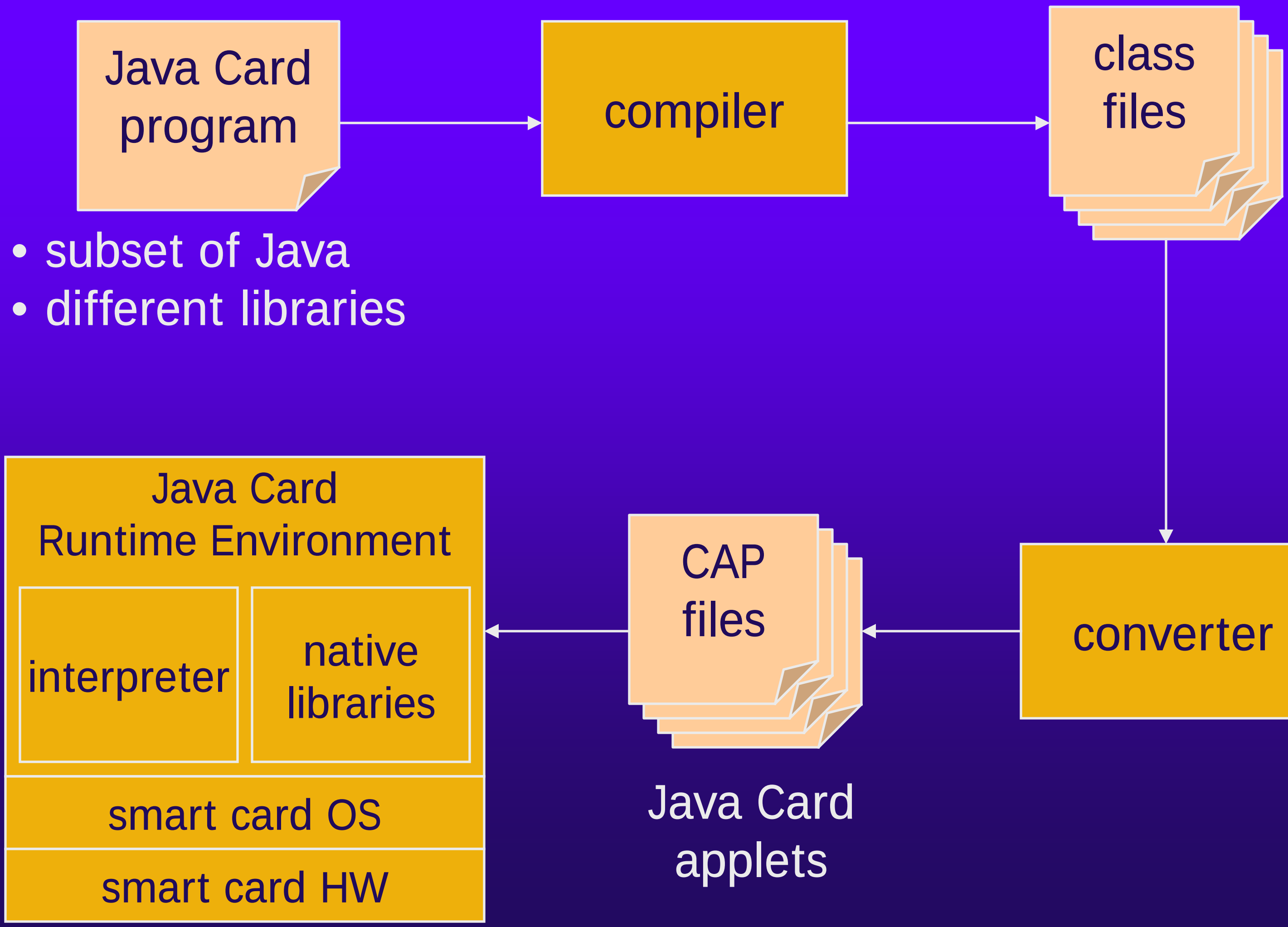
- Introduction and goals
- Type safety and security
- Bytecode verification
- Class loading
- Java Card

Smart Cards



- Security applications
  - authentication
  - encryption
  - transactions
  - ...
- High assurance is of paramount importance (also in commercial world)

Java Card Technology



Java Card program

- subset of Java
- different libraries

compiler

class files

converter

CAP files

Java Card applets

Java Card Runtime Environment

interpreter

native libraries

smart card OS

smart card HW

Java Card Security

- Applet firewall
  - isolation among applets
  - controlled communication
- Libraries supporting
  - encryption/decryption
  - signatures
  - authentication
  - ...

Our Ongoing Tasks

- High-assurance Java Card Runtime Environment (JCRE) in Specware
  - formal spec
  - refinement to code
  - can be later lifted to JVM
- Applet generators
  - provable correctness
  - reduced development time

Questions?

Backup Slides

Java vs. Bytecode

```
class C {
 int f;

 int m(int i) {
  return (f+i);
 }




 D n() {
  return (new D());
 }



}

class D { ... }
```

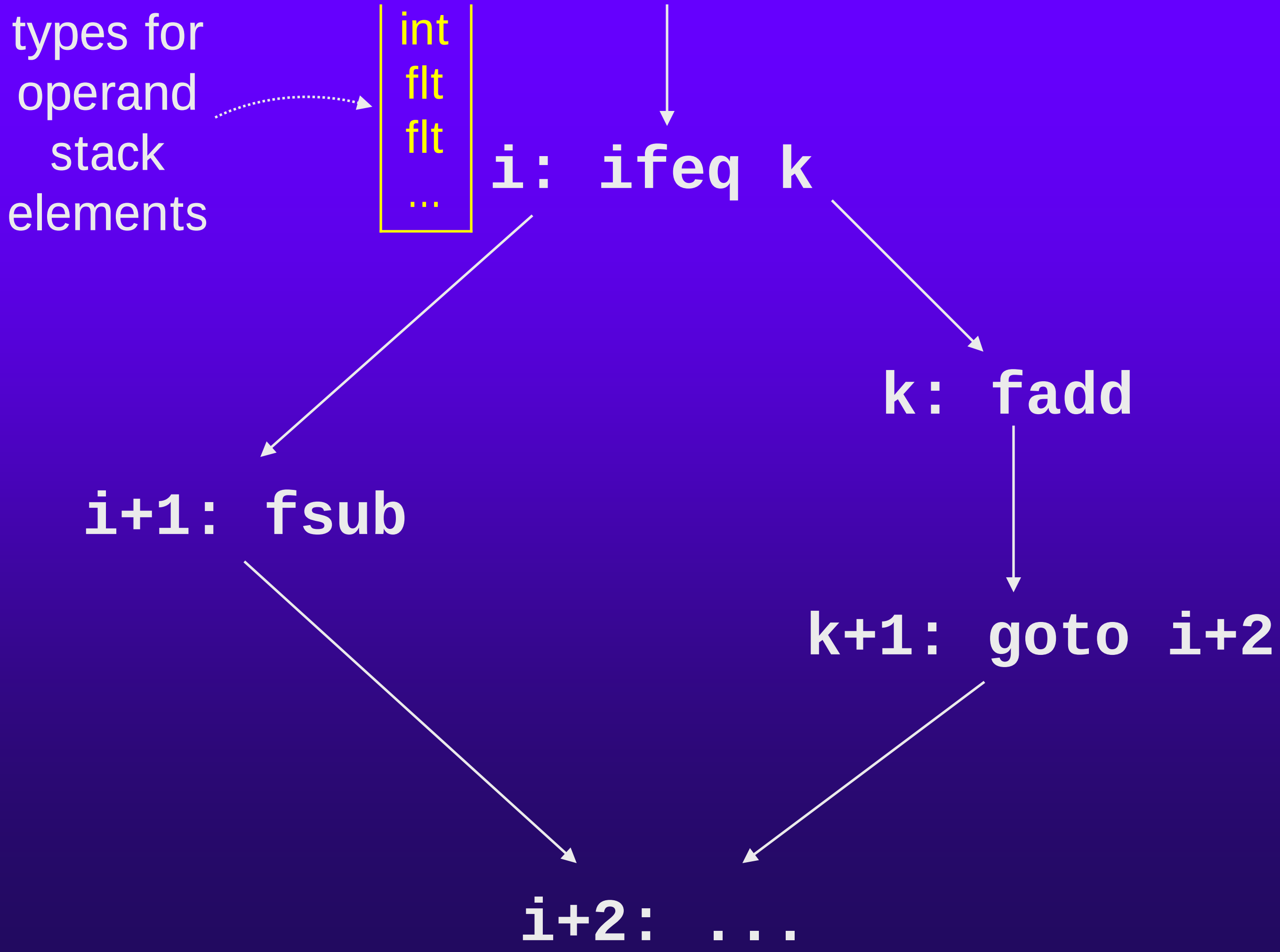
compile

```
class C {
 int f;

 int m(int) {
  aload 0
  getfield C.f
  iload 1
  iadd
  ireturn
 }
 D n() {
  new D
  dup
  invokespecial D.<init>
  areturn
 }
}

class D { ... }
```

Bytecode Verification



types for operand stack elements

int
flt
flt
…

`i: ifeq k`

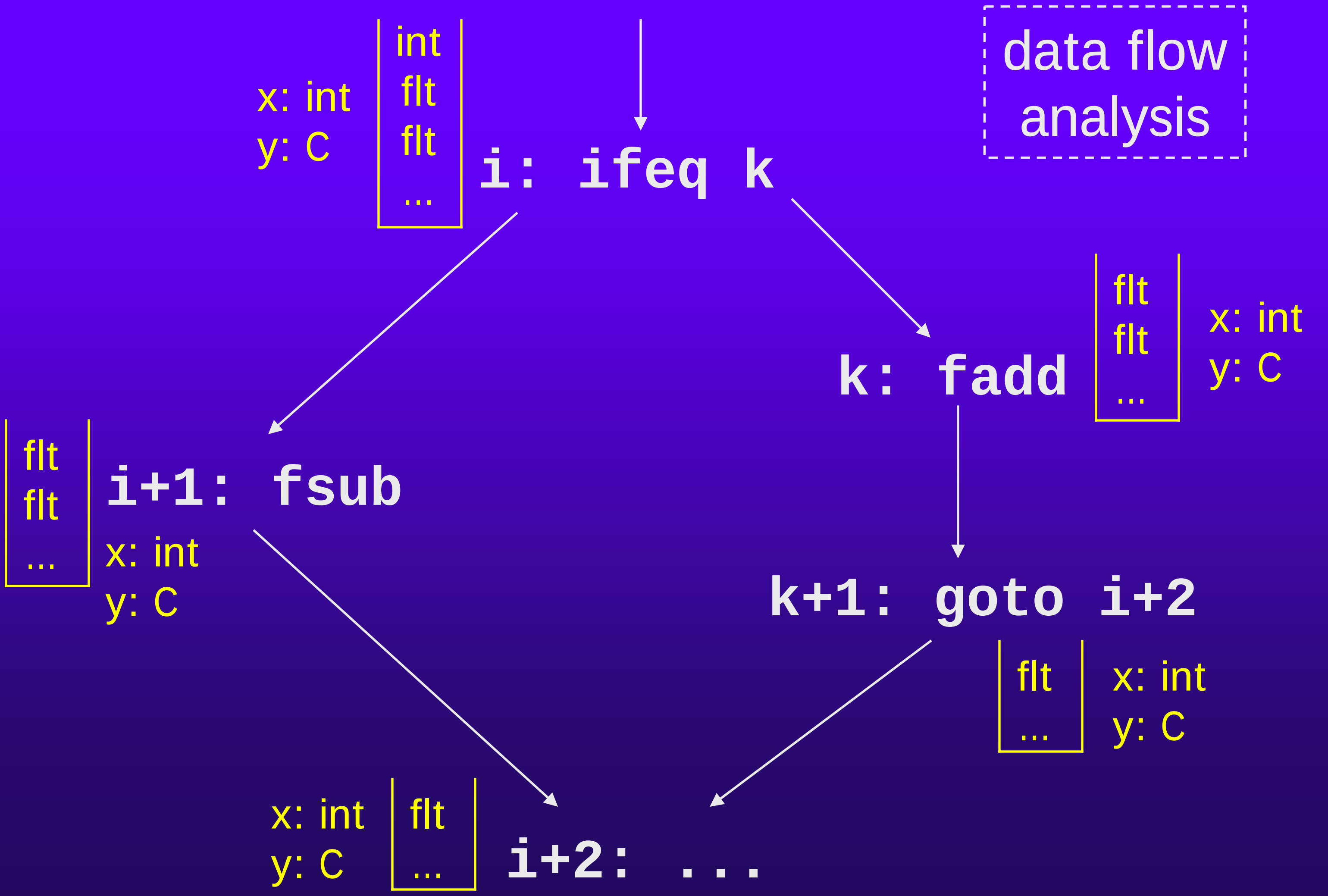
`i+1: fsub`

`k: fadd`

`k+1: goto i+2`

`i+2: ...`

Bytecode Verification



data flow analysis

x: int
y: C

int
flt
flt
...

`i: ifeq k`

`k: fadd`

flt
flt
...

x: int
y: C

flt
flt
...

`i+1: fsub`

x: int
y: C

`k+1: goto i+2`

flt
...

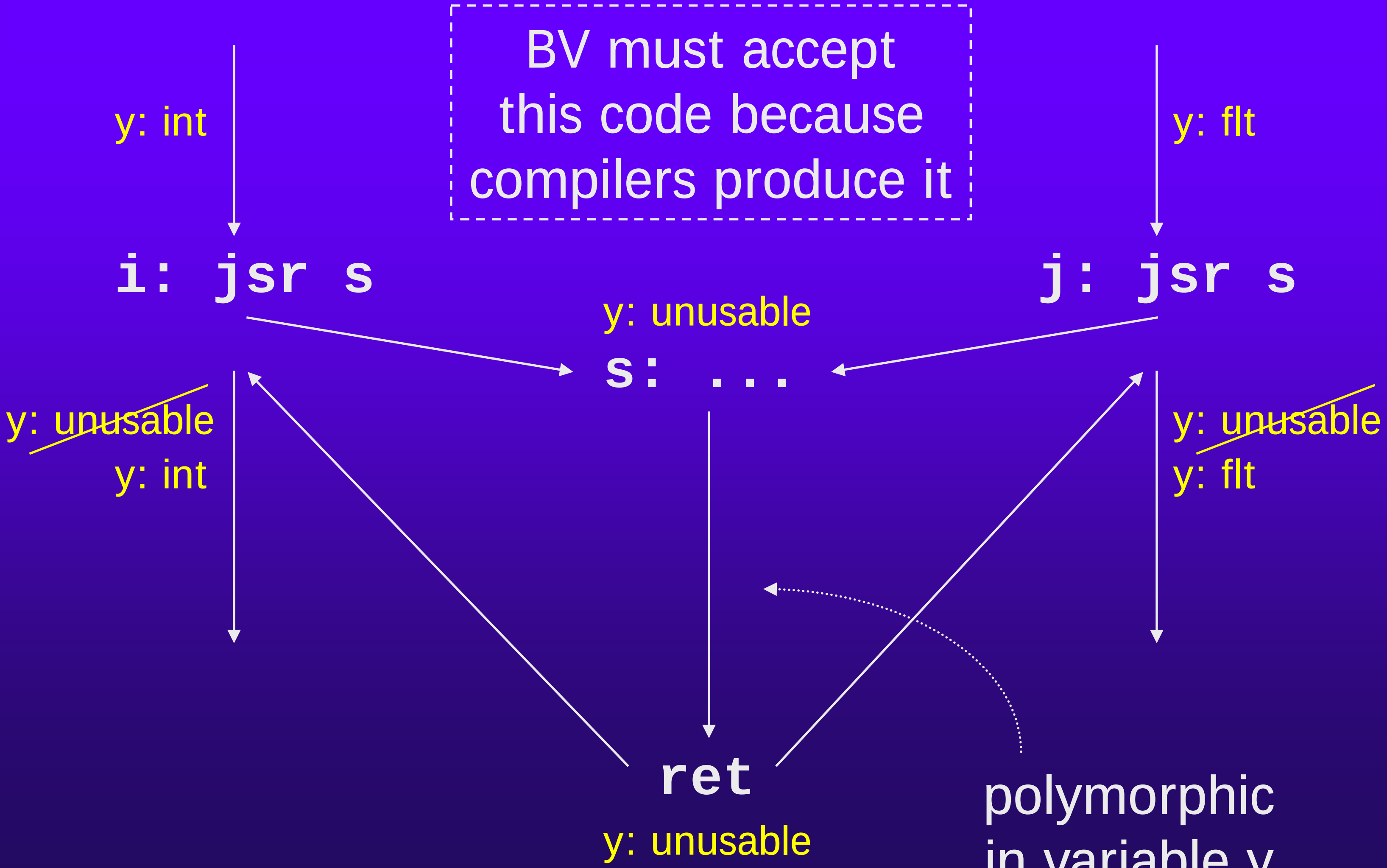
x: int
y: C

x: int
y: C

flt
...

`i+2: ...`

Polymorphic Subroutines



BV must accept this code because compilers produce it

y: int

`i: jsr s`

y: flt

`j: jsr s`

y: unusable

`s: ...`

y: ~~unusable~~
y: int

y: ~~unusable~~
y: flt

`ret`

y: unusable

polymorphic in variable y

Spec: Semilattice

```
spec SEMILATTICE

     sort L
     op join : L * L -> L

     axiom idempotence is
          join(x,x) = x
     axiom commutativity is
          join(x,y) = join(y,x)
     axiom associativity is
          join(join(x,y),z) = join(x,join(y,z))

     end-spec
```

Spec: Transfer Functions

```
spec TRANSFER-FUNCTIONS

    import SEMILATTICE

    sort TF
    op apply : TF * L -> L

    end-spec
```

Spec: Data Flow Analysis

```
spec DATA-FLOW

    import TRANSFER-FUNCTIONS

    sort Edge PP = { from : PP, tf : TF, to : PP }
    sort Prb PP = { start : PP, init : L, edges : Set (Edge PP) }

    sort Sol PP = Map (PP, L)

    op solves? : Prb PP * Sol PP -> Boolean
    ...

    op solve : Prb PP -> Sol PP
    axiom solves?(solve(p),p)

    end-spec
```