Security in programming languages

Security in programming languages

- Languages have long been related to security.
- Modern languages should contribute to security:
  - Constructs for protection (e.g., objects).
  - Techniques for static analysis,
    in particular for ensuring safety by typechecking.
  - A tractable theory, with sophisticated methods.
- Several security techniques rely on language ideas, with static and dynamic checks.

See Morris's "Protection in Programming Languages".

A class with a secret field

```
class C {
    // the field
    private int x;
    // a constructor
    public C(int v) { x = v; }
}

// two instances of C
C c1 = new C(17);
C c2 = new C(28);
```

- A possible conjecture: *Any two instances of this class are observationally equivalent (that is, they cannot be distinguished within the language).*
- More realistic examples use constructs similarly.
- Objects are unforgeable. E.g., integers cannot be cast into objects.

Mediated access [example from A. Kennedy]

```
class Widget {// No checking of argument
  virtual void Operation(string s) {…};
}
class SecureWidget : Widget {
  // Validate argument and pass on
  // Could also authenticate the caller
  override void Operation(string s) {
    Validate(s);
    base.Operation(s);
  }
}
…
SecureWidget sw = new SecureWidget();
sw.Operation(“Nice string”);
// Can’t avoid validation of argument
```

Some application areas

Avoiding or detecting implementation flaws

- Various static analyses (e.g., for C programs).
- High-level languages with automatic memory management (e.g., Java).
- Typed assembly and intermediate languages; proof-carrying code.

Using language constructs to embody policies

- Data abstraction for protecting private state.
- Objects for capabilities.
- Method calls enriched with security semantics (as in stack inspection).
- Information-flow control (e.g., in Jif).

Other aspects of language-based security

- Secure coding practices.
- Signed code (e.g., with “strong names”).
- Libraries for cryptography, authentication, . . .
- Support for inline reference monitors.
- Code obfuscation.

Firstly, the notation should be designed to reduce as far as possible the scope for coding error; or at least to guarantee that such errors can be detected by a compiler, before the program even begins to run. Certain programming errors cannot always be detected in this way, and must be cheaply detectable at run time; in no case can they be allowed to give rise to machine- or implementation-dependent effects, which are inexplicable in terms of the language itself. This is a criterion to which I give the name ***security****.*

Hoare (1973)

The security flaw reported this week in E-mail programs produced by two highly respected software companies points to an industrywide problem – the danger of programming languages whose greatest strength is also their greatest weakness.

More modern programming languages, like the Java language developed by Sun Microsystems, have built-in safeguards that prevent programmers from making many common types of errors that could result in security loopholes.

The New York Times (1998)

Types

- A program variable can assume a range of values during the execution of a program.
- An upper bound of such a range is called a type of the variable:
  - A variable of type “bool” is supposed to assume only boolean values.
  - If x has type “bool” then “not(x)” has a sensible meaning during every run of the program.
- An important application of type systems is to prevent execution errors.

Typed vs. untyped languages

Untyped languages:

- The language does not restrict the range of values for a given variable.
- Operations might be applied to inappropriate arguments.
  The behavior in such cases might be unspecified.

Typed languages:

- Variables can be assigned (non-trivial) types.
- Types might or might not appear in programs.

Trapped vs. untrapped errors

Trapped errors:

- Trapped errors give rise to well-specified behavior.
  - E.g., division by zero.
- Even languages with powerful type systems permit trapped errors.

Untrapped errors:

- Untrapped errors lead to unspecified behavior which depends on machine state.
  - E.g., accessing past the end of an array.
  - E.g., jumping to an address in the data segment.

Safe languages

- A program is *safe* if it does not cause untrapped errors.
- Languages where all programs are safe are *safe languages*.
  - Some languages with types are not safe ("weakly typed languages").
  - A compromise is the isolation of unsafe code (as in Cedar).

A typical theorem

A computation state consists of:

- values for program variables,
- a program counter,
- . . .

or the special state ***wrong*** to represent untrapped errors.

A computation step is a state transition, written $s \rightarrow t$.

Suppose that P is a program that typechecks, and that we run P from initial state s_0 and:

- s_0 is not ***wrong***,
- $s_0 \rightarrow s_1 \rightarrow . . . \rightarrow s_{n-1} \rightarrow s_n$.

Then s_n is not ***wrong***.

Alternative formulation (modelling errors by stuck states)

A computation state consists of:

- values for program variables,
- a program counter,
- . . .

(but without a special state ***wrong***).

A computation step is a state transition, written $s \rightarrow t$.

Suppose that P is a program that typechecks, and that we run P from initial state s_0 and:

- $s_0 \rightarrow s_1 \rightarrow . . . \rightarrow s_{n-1} \rightarrow s_n$,
- there is no t such that $s_n \rightarrow t$.

Then in s_n the program counter points to a halt instruction in P.

Security with safe languages

Safe languages permit:

- predictable behavior (despite sharing),
- unforgeable capabilities,
- mediation guarantees,

and (in comparison with hardware protection)

- portability,
- often adequate efficiency,
- rich interfaces.

But safety does not automatically imply security.

Beyond safety

Safety is a foundation. We may also have higher-level objectives, e.g., secrecy properties.

In a typical scenario, a host runs some foreign code and wants some security guarantees.

- Safety clearly helps.

But there are other scenarios:

- The foreign code may want to some guarantees, e.g., no reverse engineering.
- Two pieces of foreign code may coexist.

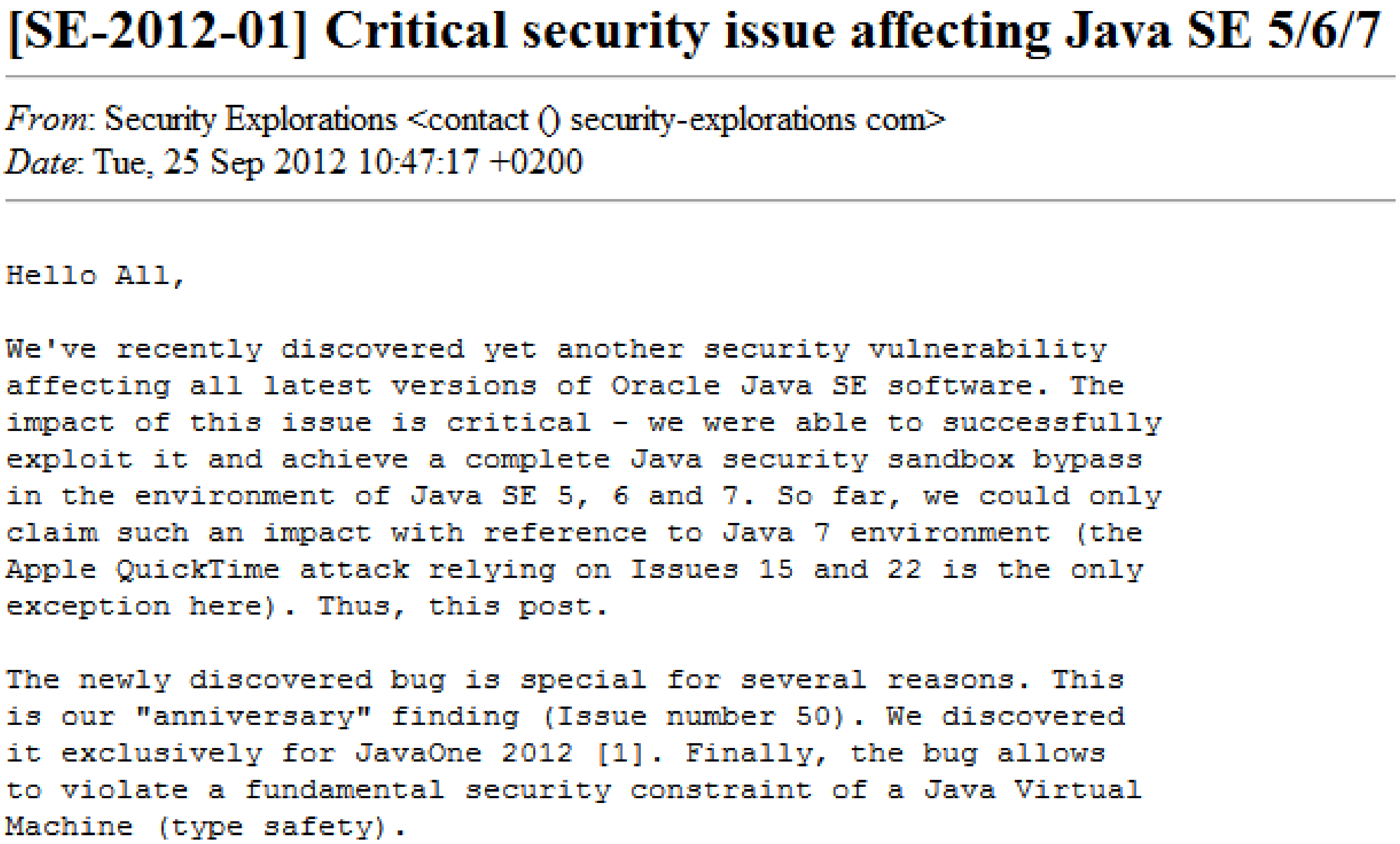
Caveats

Mismatch in characteristics:

- Security requires simplicity and minimality.
- Common programming languages are complex.

Mismatch in scope:

- Language descriptions rarely specify security. Implementations may or may not be secure.
- Security is a property of systems (not languages). Systems typically include much security machinery beyond what is given in language definitions.



[SE-2012-01] Critical security issue affecting Java SE 5/6/7

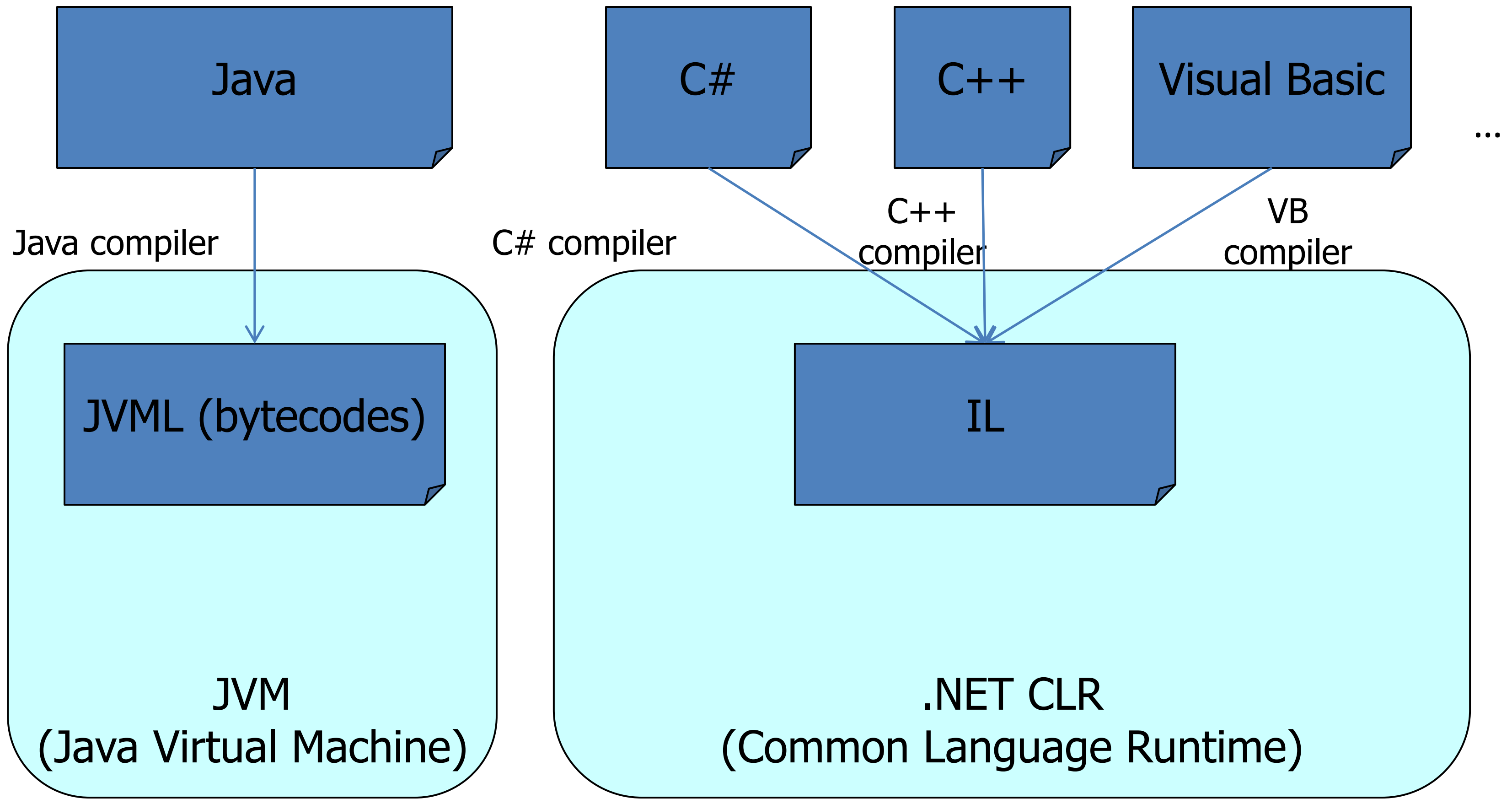
From: Security Explorations <contact () security-explorations com>
Date: Tue, 25 Sep 2012 10:47:17 +0200

Hello All,

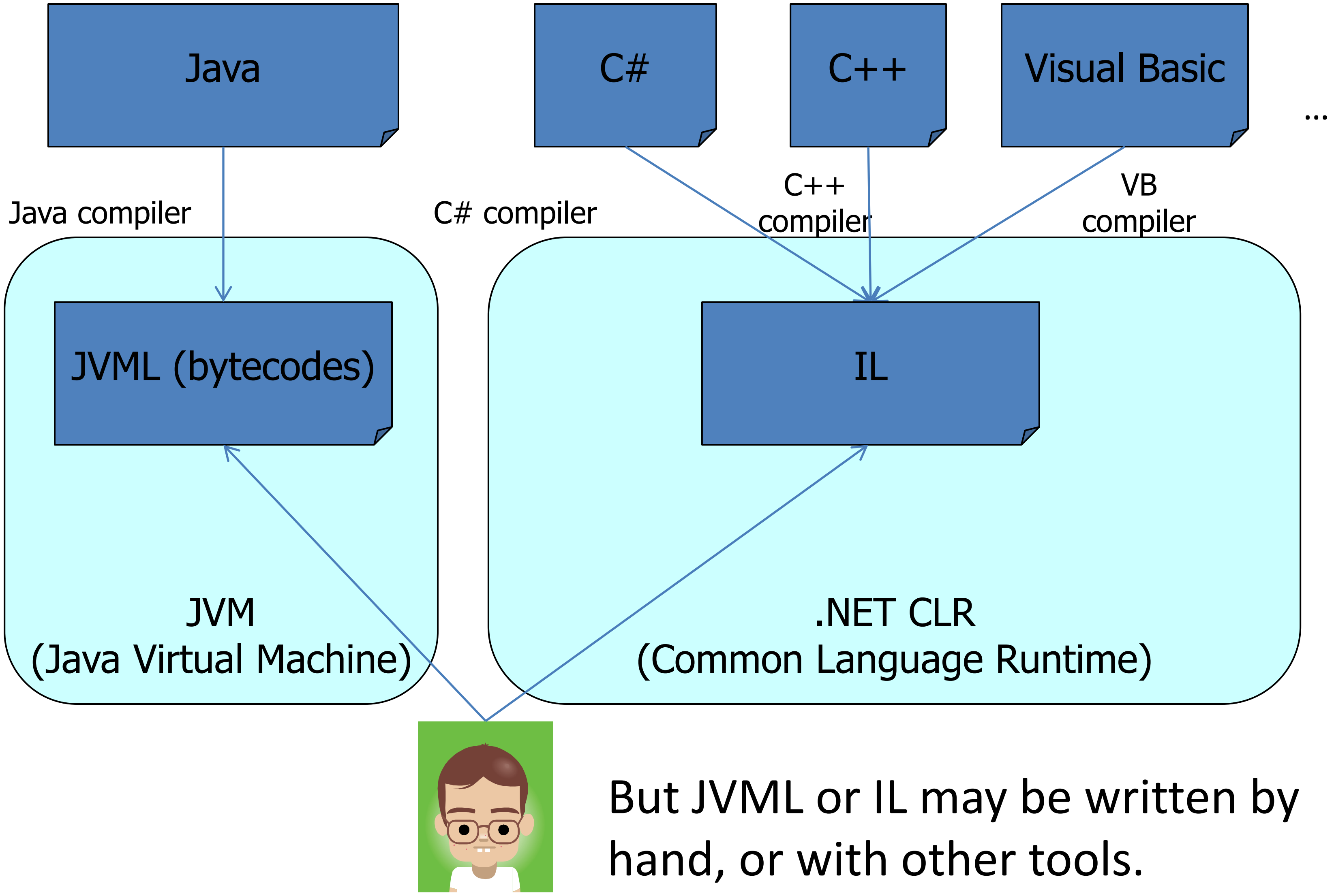
We've recently discovered yet another security vulnerability
affecting all latest versions of Oracle Java SE software. The
impact of this issue is critical - we were able to successfully
exploit it and achieve a complete Java security sandbox bypass
in the environment of Java SE 5, 6 and 7. So far, we could only
claim such an impact with reference to Java 7 environment (the
Apple QuickTime attack relying on Issues 15 and 22 is the only
exception here). Thus, this post.

The newly discovered bug is special for several reasons. This
is our "anniversary" finding (Issue number 50). We discovered
it exclusively for JavaOne 2012 [1]. Finally, the bug allows
to violate a fundamental security constraint of a Java Virtual
Machine (type safety).

“Secure” programming platforms

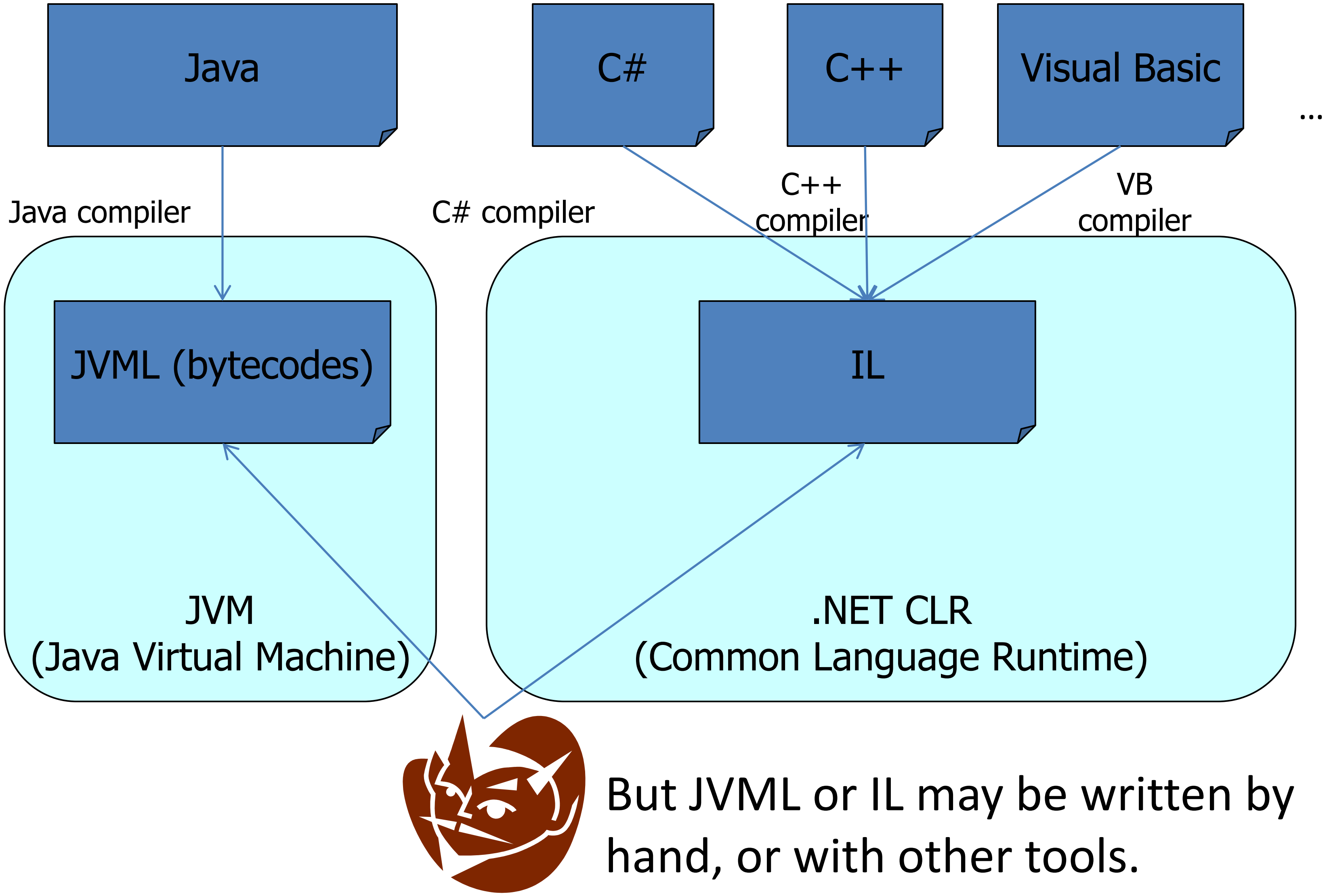


“Secure” programming platforms



But JVML or IL may be written by hand, or with other tools.

"Secure" programming platforms



But JVML or IL may be written by hand, or with other tools.

Verifying intermediate- and low-level code

Intermediate- and low-level code may not have been produced by the intended compilers.

- So typechecking source code is not enough!
- Various systems (e.g., JVMs and .NET CLR) include verifiers for lower-level code.
- These verifiers are basically typecheckers.

(The goal is to protect the host systems, not the untrusted code, which anyway is typically subject to lower-level attacks.)

Another Java class with a secret

```
class C {
    private int x;
    public void set_x(int v) {
        this.x = v;
    };
}
```

- A possible conjecture: Any two instances of this class cannot be distinguished within the language.

Secrecy is preserved by translation (sometimes)

- The same class at the bytecode level:

```
class C {
    private int x;
    public void set_x(int) {
      .framelimits locals = 2, stack = 2;
      aload_0;       // load this
      iload_1;       // load v
      putfield x;    // set x
    };
}
```

- Bytecode verification is required.

Verification

In bytecode verification, at each program point:

- the stack gets a height,
- each stack location gets a type,
- each local variable gets a type.

Various checks are then performed, e.g.,

- the stack never overflows or underflows,
- operations applied to operands of appropriate types,
- objects are not used before initialization,
- returns lead back to jump sites.

Some checks are left for runtime,

- e.g., array-bounds checks.

A miniature verifier

- We will give a formal treatment of a tiny language MicroIL and a verifier for it.
  - MicroIL is a fragment of the language of "A type system for Java bytecode subroutines" http://dl.acm.org/citation.cfm?doid=314602.314606
  - It resembles other lower-level languages.
- The goal is to explain that lower-level typechecking can be specified and analyzed precisely, not to cover every feature.

MicroIL programs

A program is a sequence of instructions:

$$\begin{array}{rl} \textit{instruction} ::= & \texttt{inc} \\ & |\,\texttt{pop} \\ & |\,\texttt{push0} \\ & |\,\texttt{load}\ x \\ & |\,\texttt{store}\ x \\ & |\,\texttt{if}\ L \\ & |\,\texttt{halt} \end{array}$$

where x ranges over Var (the set of variables),

and L ranges over Addr (the set of addresses).

MicroIL states

A state is a triple $\langle pc, f, s\rangle$ where:

- pc is an address,
- f maps variables to values,
- s is a stack of values.

MicroIL semantics

Rules for $P \vdash \langle pc, f, s \rangle \rightarrow \langle pc', f', s' \rangle$

$$\frac{P[pc] = \texttt{inc}}{P \vdash \langle pc, f, n \cdot s \rangle \rightarrow \langle pc+1, f, (n+1) \cdot s \rangle}$$

$$\frac{P[pc] = \texttt{pop}}{P \vdash \langle pc, f, v \cdot s \rangle \rightarrow \langle pc+1, f, s \rangle}$$

$$\frac{P[pc] = \texttt{load}\ x}{P \vdash \langle pc, f, s \rangle \rightarrow \langle pc+1, f, f[x] \cdot s \rangle}$$

$$\frac{P[pc] = \texttt{push0}}{P \vdash \langle pc, f, s \rangle \rightarrow \langle pc+1, f, 0 \cdot s \rangle}$$

$$\frac{P[pc] = \texttt{if}\ L}{P \vdash \langle pc, f, 0 \cdot s \rangle \rightarrow \langle pc+1, f, s \rangle}$$

$$\frac{P[pc] = \texttt{store}\ x}{P \vdash \langle pc, f, v \cdot s \rangle \rightarrow \langle pc+1, f[x \mapsto v], s \rangle}$$

$$\frac{\begin{array}{c} P[pc] = \texttt{if}\ L \\ n \neq 0 \end{array}}{P \vdash \langle pc, f, n \cdot s \rangle \rightarrow \langle L, f, s \rangle}$$

Verifying MicroIL

The typing rules of MicroIL should prevent:

- type errors,
- operand stack overflow or underflow,
- wild jumps,

but they should allow local typing
(different types for the same variable or stack location at different points).

MicroIL typing rules

Rules for $F, S \vdash P$ means that P is well-typed, with types given by F and S

where

- $F_{pc}[x]$ is the type of variable x at point pc (possibly undefined),
- S_{pc} is the type of the operand stack (i.e., one type for each stack slot) at point pc.

Basic types for values: Top, Int, …

MicroIL typing rules (cont.)

The top-level rule is:

$$\frac{\begin{array}{c}\forall x \in \mathit{Var}.\, F_1[x] = \mathsf{Top} \\ S_1 = \epsilon \\ \forall i \in \mathit{Dom}(P).\, F, S, i \vdash P\end{array}}{F, S \vdash P}$$

where:

- Top is the biggest type,
- ϵ is the empty stack,
- $F, S, i \vdash P$ is a local typing check for program point i.

MicroIL typing rules (cont.)

One rule for each instruction, such as:

$$\frac{\begin{array}{c} P[i] = \texttt{inc} \\ F_{i+1} = F_i \\ S_{i+1} = S_i = \mathsf{Int} \cdot \alpha \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{load}\ x \\ x \in \mathit{Dom}(F_i) \\ F_{i+1} = F_i \\ S_{i+1} = F_i[x] \cdot S_i \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P}$$

$$\frac{\begin{array}{c} P[i] = \texttt{push0} \\ F_{i+1} = F_i \\ S_{i+1} = \textsc{Int} \cdot S_i \\ i+1 \in \mathit{Dom}(P) \end{array}}{F, S, i \vdash P}$$

$$\frac{P[i] = \texttt{halt}}{F, S, i \vdash P}$$

A tiny example

If P is the program load x · load x · halt
then P is well-typed, with F and S given by:

- $F_1[x]$ = Top, S_1 = ε
- $F_2[x]$ = Top, S_2 = Top · ε
- $F_3[x]$ = Top, S_3 = Top · Top · ε

A theorem

Given a program P, F, and S such that $F, S \vdash P$,

if $P \vdash \langle 1, f_0, \epsilon \rangle \rightarrow^* \langle pc, f, s \rangle$

and there is no pc', f', s'

such that $P \vdash \langle pc, f, s \rangle \rightarrow \langle pc', f', s' \rangle$

then $s : S_{pc}$

and $P[pc] = \mathtt{halt}$.

$\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$.

In other words, if a program typechecks, then it makes progress until it halts (without uncaught errors).

Another Java class with a secret field?

```
class D {
    class E {
        private int y = x;
    };
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
}
```

- E is an inner class.

An accessor can break secrecy

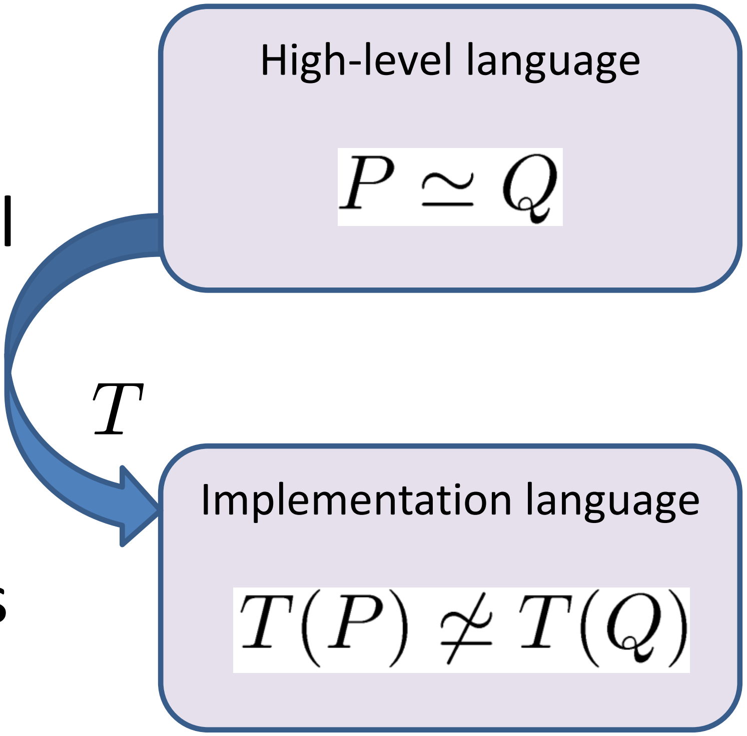
```
class D {
    private int x;
    public void set_x(int v) {
      this.x = v;
    };
    static int get_x(D d) {
      return d.x;
    };
}

class E {
    ... get_x ...
}
```

This is the way inner classes were originally desugared.
Other techniques have been tried since then.

Other examples

There are many more examples, for Java, C#, and other languages.

- In each case, some observational equivalence that holds in the source language does not hold in implementations.
- We may say that the translations are not ***fully abstract***.
- Typechecking helps, but it does not suffice.



Mediated access, revisited

```
class Widget {// No checking of argument
  virtual void Operation(string s) {…};
}
class SecureWidget : Widget {
  // Validate argument and pass on
  // Could also authenticate the caller
  override void Operation(string s) {
    Validate(s);
    base.Operation(s);
```

```
// In IL (pre-2.0), make a direct call
// on the superclass:
ldloc sw
ldstr “Invalid string”
call void Widget::Operation(string)
```

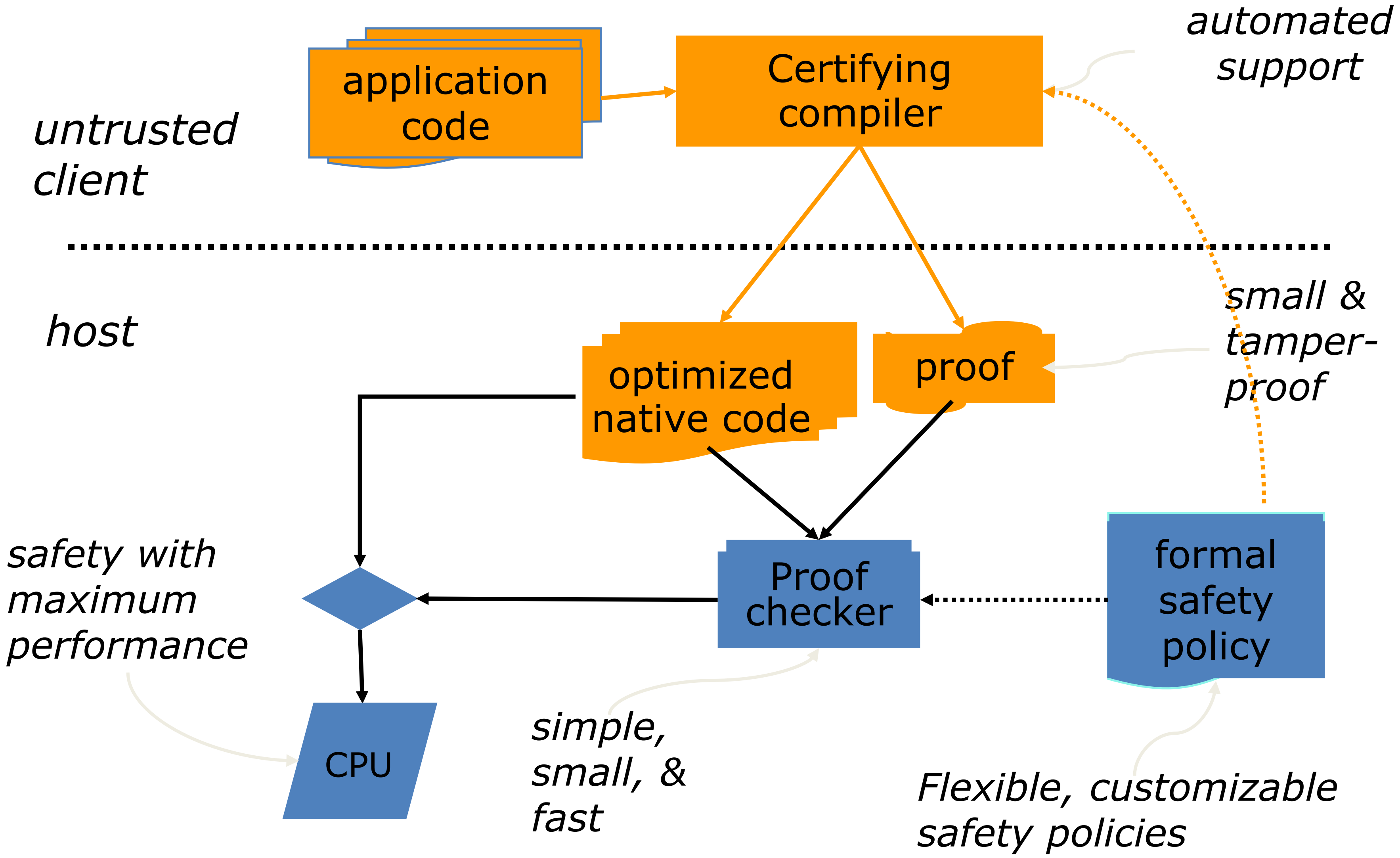
Alternatives

- One may ignore the security of translations
  - when low-level code is signed by a trusted party,
  - if one analyzes low-level code.

  These alternatives are not always satisfactory.

- In other cases, translations should preserve at least some security properties; for example:
  - limited versions of full abstraction (e.g., for certain programming idioms),
  - the secrecy of pieces of data labelled as secret,
  - fundamental guarantees about control flow.

Proof-Carrying Code (PCC)



Abstractions and security

Abstractions are common in computing, e.g.:

- function calls,
- objects with private components,
- secure channels.

Clever implementation techniques abound too:

- stacks,
- static and dynamic access checks,
- cryptography.

Abstractions and security

Abstractions are common in computing, e.g.:

- function calls,
- objects with private components,
- secure channels.

Clever implementation techniques abound too:

- stacks,
- static and dynamic access checks,
- cryptography.

Implementations often need to work in interaction with (malicious?) systems that do not use the abstractions.

Abstractions and security

Abstractions are common in computing, e.g.:

- function calls,
- objects with private components,
- secure channels.

Clever implementation techniques abound too:

- stacks,
- static and dynamic access checks,
- cryptography.

Implementations often need to work in interaction with (malicious?) systems that do not use the abstractions.

This holds even for low-level code, and ideas originally developed in high-level languages are useful there too.

Reading

- Morris' "Protection in programming languages"
  http://dl.acm.org/citation.cfm?id=361937&CFID=162158828&CFTOKEN=38362855
- Kennedy's "Securing the .NET Programming Model"
  http://research.microsoft.com/en-us/um/people/akenn/sec/appsem-tcs.pdf