

Security in programming languages

Security features of languages

Languages have long been related to security (see Morris).

Modern languages should contribute to security:

- Constructs for protection (e.g., objects).
- Predictable behavior.
- Techniques for static analysis, in particular for ensuring safety by typechecking.
- Precise definitions (e.g., of type soundness).
- A tractable theory, with sophisticated methods.

Static analysis for security

Recent applications:

- Bytecode verification
- Proof-carrying code
- Typed assembly language
- Buffer-overflow detection for C
- Information-flow security by typing
 - in programming
 - in protocol analysis

Problems

Security requires simplicity and minimality.

Common programming languages are complex.

Common programming-language designs and implementations are not structured for security.

Each programming language typically comes with standard libraries (with their own flaws).

Language descriptions rarely specify security.

Implementations may or may not be secure.

Outline

Safety and security

Other aspects of security in languages

Java background

Access control in Java

Low-level security: bytecode verification

A little more on information-flow analysis

Reading

A paper by Morris:

- “Protection in programming languages”
in the ACM digital library.

Other papers will be mentioned later.

Safety and security

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.

Some CERT statistics

according to S. Bellovin in 1998

- (a) 85% of all CERT advisories represent problems that crypto can't fix, and
- (b) 30-50% of recent security hole reports involve buffer overflow.

CNET on the “I Love You” virus (2000)

” You can say a lot of things . . . how Java’s not good, and you can say JavaScript has a lot of flaws,” Zboray said. ” But the security posture from which they were designed was the right posture. The security posture from which ActiveX and VBScript were designed is the wrong posture.”

For its part, Microsoft attributes the ongoing security issues not so much to inherent problems with Visual Basic script and its macro language, but to bad people misusing good software.

Tony Hoare said (1973)

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*.

We call it [safety](#) or [memory-safety](#) instead.

Reading on typed languages and safety

A paper by Cardelli:

- "Type systems"

at www.luca.demon.co.uk/.

(Suggested if you need background, not required.)

Safe programs

A safe program is one that cannot corrupt the run-time system.

- An error may cause a computation to abort, or to give the wrong answer, but not a system crash.
- Safe programs may share an address space with limited danger.

A program that is allowed to access an arbitrary address can corrupt an address space arbitrarily.

For example, after turning an integer into a reference, the program may read or clobber the data of any other program in its address space.

Errors

No error:

$$2 \times 2 \rightarrow 4$$

Error, but caught:

$$1000/0 \rightarrow \text{“divide by 0 exception”}$$

Uncaught error:

$$1000 + \text{“hello”} \rightarrow r$$

where r depends on the representation of numbers and strings, and some parts of memory may be corrupted!

Representing errors

We are not going to discuss caught errors further.

There are two common ways to represent uncaught errors:

- $1000 + \text{"hello"} \rightarrow \text{wrong}$

where `wrong` is a new constant.

- $1000 + \text{"hello"}$ is stuck (cannot make progress):
there is no r such that $1000 + \text{"hello"} \rightarrow r$.

They are essentially equivalent.

We adopt the latter.

Safe languages

In some languages, all bets are off (BCPL, assembly language).

Others aim to allow only safe programs (pure Lisp, pure Java).

A compromise is the isolation of unsafe code (Cedar, Modula-3, Java with native methods, C#).

A typical theorem about a safe language

Suppose that P is a syntactically correct program and typechecks.

A computation state consists of:

values for program variables,

a program counter,

...

Suppose that we run P from initial state s_0 , and:

- $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_{n-1} \rightarrow s_n$
- there exists no state t such that $s_n \rightarrow t$

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

Security with safe languages

Safe languages permit:

- predictable behavior,
- unforgeable capabilities,
- mediation guarantees,

and (in comparison with hardware protection)

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

Safety does not automatically imply security.

Other aspects of security in languages

Beyond (or above) safety

Safety is a foundation.

We may have higher-level security objectives,
e.g., secrecy properties.

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

But there are other scenarios:

- the foreign code may want to some guarantees,
e.g., no reverse engineering,
- two pieces of foreign code may need to coexist.

Other aspects of security

- Access control (for example with stack inspection).
- Signed code.
- Proper linking.
- Information-flow analysis.
- Libraries for cryptography, authentication,
- Secure coding practices.

All of these are present in Java or close relatives.

Java background

Which Java?

The Java language evolves.

Several implementations exist, including several virtual machines (e.g., for small devices).

C# and the CLR (Microsoft's Common Language Runtime) have many similarities to Java and the JVM.

These lectures are based on a standard Java.

Details don't necessarily apply to variants, but the general ideas do.

Java generalities

Java has:

- C-like syntax
- classes
- objects, generated from classes
- object types, called interfaces
- single inheritance for classes
- multiple inheritance for interfaces
- static typing (with safety goals)
- dynamic type tests (via instanceof)

Java generalities (cont.)

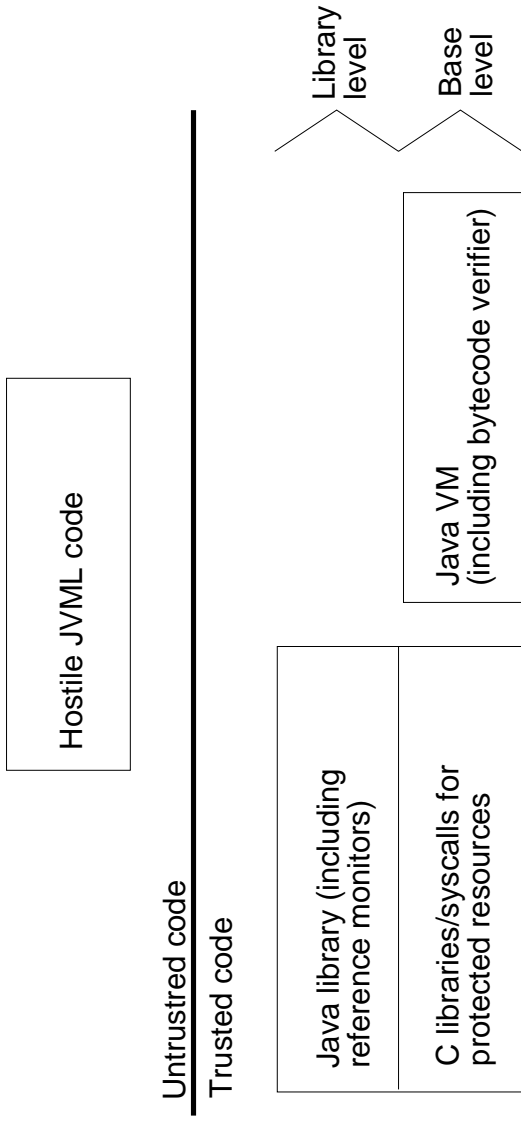
Java also has:

- typed exceptions
- arrays
- not many other types!
- garbage collection
- concurrency, with threads
- synchronization
- packages with visibility rules

The Java Virtual Machine

Common Java compilers produce code in an intermediate language that we call JVMIL.

The Java Virtual Machine (JVM) executes JVMIL code.



The JVM is part of web browsers and other applications.

A class in Java

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

More substantial and realistic examples can use objects and privacy in similar ways.

- Object references are unforgeable capabilities.
- Access to private components is mediated.

(Package protection is similar to privacy.)

Security objectives

Language-based security (e.g., privacy) can be important for protection **against** mobile objects.

- Java libraries interact with mobile code in JVMML.
- Inter-applet communication may remain within JVMML.

It is less meaningful for protection **of** mobile objects.

- Mobile objects on untrusted machines are subject to lower-level attacks.

Access control in Java

Access control: the sandbox and beyond

The sandbox policy:

- Local code has the full power of its owner.
 - Foreign code has very limited rights:
 - no direct use of files,
 - network connections only back to the code's origin,
- :

Enforcement:

- A security manager is associated with code when the code is loaded.
- The security manager serves as a reference monitor for requests from the code.

The sandbox is generally too inflexible.

Stack inspection

Components with a variety of origins, more or less trusted, share the runtime, and may call one another.

Access to protected resources is expressed in terms of permissions, such as “may perform screen I/O” .

A configurable policy decides what permissions are available to each component given evidence of its origin.

Before execution, each function or method body receives an annotation that determines its permissions.

Stack inspection (cont.)

When code tries to access a protected resource, it does not suffice that it have the appropriate permission:

- Basically, all code traversed on the thread stack must have the appropriate permission.
- Trusted code may invoke a primitive (`BeginPrivilege`) to override the inspection of its callers and hence to take responsibility.

The purpose of inspecting the stack is to prevent the “confused deputy” problem.

Examples

Suppose that there is a library function $f(d,q)$, where:

- d is a directory,
- q is a query,
- $f(d,q)$ returns the results of running q on the files in d .

When f is called by some function g , both should have the permission to look at the files in d .

f should not call `BeginPrivilege`.
(Otherwise, it is a “confused deputy” .)

Examples (cont.)

Suppose that there is a library function $f(q)$, where:

- q is a query,
- $f(q)$ returns the results of running q on the public Web,
- f updates a log file with every query,
- f may look in cache files and use temporary files.

When f is called by some function g , only f need have the permission to touch log, cache, and temporary files.

f should call `BeginPrivilege`.

Criticisms of stack inspection

Does it achieve real security? for what policy?

Some constructs (e.g., threads, method delegation) require careful treatment.

A standard formulation of stack inspection is tied to a particular stack implementation.

⇒ It rules out or complicates optimizations.

Checking privileges can be expensive.

Further examples

Written in C#.

Illustrate limitations of stack inspection.

From joint work with Cédric Fournet.

```
// Trusted : static permissions contain all permissions .  
public class File { ...  
    public static void Delete(string s) {  
        FileIOPermission p = new FileIOPermission(s ...));  
        p.Demand();  
        Win32.Delete(s);  
    }  
}
```

```
// Mostly untrusted : static permissions don't
// contain any FileIOPermission.
class BadApplet {
    public static void Main() {
        NaiveLibrary.CleanUp("..\\password");
    }
}
// Trusted : static permissions contain all permissions .
public class NaiveLibrary {
    public static void CleanUp(string s) {
        File.Delete(s);
    }
}
```

```
// Trusted : static permissions contain all permissions .
class NaiveProgram {
    public static void Main() {
        string s = BadPlugin.TempFile();
        File.Delete(s);
    }
}
// Mostly untrusted : static permissions don't
// contain any FileIOPermission .
public class BadPlugin {
    public static string TempFile() {
        return "..\\password";
    }
}
```

```
public sealed class Task {
    private string s;
    public Task(string s) { this.s = s; }
    public void Start () { File.Delete(s); }
}
public class Untrusted {
    // The following declarative attribute removes
    // all FileIOPermissions for this method.
    [ FileIOPermissionAttribute
      ( SecurityAction .Deny, Unrestricted =true )]
    public static Task applet () {
        return new Task( "..\\password" );
    }
}
class Program {
    static void Main() {
        Untrusted . applet (). Start ();
    }
}
```

Reading on stack inspection and more

Java's documentation,

at: java.sun.com/j2se/1.4/docs/guide/security/

and in particular

java.sun.com/j2se/1.4/docs/guide/security/permissions.html

Papers from the Princeton SIP group,

at www.cs.princeton.edu/sip/pub/index.php3.

A paper by Fournet and Gordon:

- "Stack inspection: theory and variants",
at research.microsoft.com/~fournet/biblio.htm.

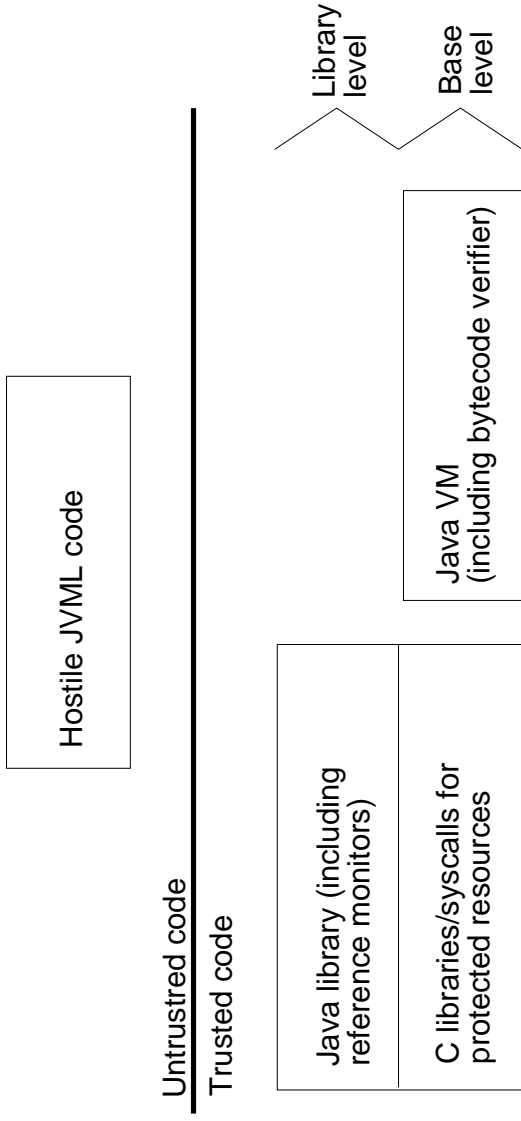
(Suggested, not required.)

Low-level security: bytecode verification

The Java Virtual Machine (reminder)

Common Java compilers produce code in an intermediate language that we call JVMIL.

The Java Virtual Machine (JVM) executes JVMIL code.



The JVM is part of web browsers and other applications.

A class in Java

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

More substantial and realistic examples can use objects and privacy in similar ways.

- Object references are unforgeable capabilities.
- Access to private components is mediated.

(Package protection is similar to privacy.)

The same class in JVMIL

```
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

Bytecode verification: outline

Part I: basics of bytecode verification in Java

Part II: bytecode verification in a tiny language

Part III: some advanced features

Part IV: a limitation

Bytecode verification, Part I

The Java bytecode verifier

The Java bytecode verifier is a popular piece of language-based security infrastructure.

- It is perhaps the most widespread typechecker.
- It plays an important part in the Java security story (and concerns).
- It has a 20-page informal specification.
- Sun's original implementation consisted of ~ 4000 lines of code.

Errors in JVMML programs

For security, one cannot ignore errors:

- type errors,
- operand stack overflow or underflow,
- access control violations,
- reading of uninitialized variables,
- use of uninitialized objects,
- wild jumps.

Otherwise, for example, the qualifier **private** is worth little or nothing.

Bytecode verification: general strategy

The bytecode verifier helps prevent errors by checking untrusted JVMIL code before execution.

In bytecode verification, at each program point:

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

Some checks are left for runtime,

e.g., array-bounds checks.

Understanding the bytecode verifier

Our goals are to understand:

- bytecode verification in general,
- Sun's specification and implementation.

Some observations about Sun's bytecode verifier:

- Its specification mixes "what" and "how" .
- Its implementation and specification disagree.
- It may well be all right.
- It is very interesting, maybe too interesting.
- It was not designed for ease of analysis.

The specification of Java discusses JVMIL (!).

Verification ensures that the binary representation of a class or interface is structurally correct. For example, it checks that every instruction has a valid operation code; that every branch instruction branches to the start of some other instruction, rather than into the middle of an instruction; that every method is provided with a structurally correct signature; and that every instruction obeys the type discipline of the Java language.

When does JVMIL code “obey the type discipline of the Java language” ?

Inference vs. checking

The original bytecode verifiers rely on type inference.

- Verification was an afterthought.
- Class files did not carry type information for stack locations and local variables, originally.
- Inference reduces changes and space requirements.

But inference can be complex and expensive.

More recent verifiers rely on more checking
e.g., for J2ME CLDC (Connected Limited Device
Configuration) for phones and the like.

A better bytecode verifier?

Type theory suggests a clearer approach to bytecode verification:

- specification = a type-soundness property,
- an abstract implementation = typing rules,
- a concrete implementation = a typechecker.

A better bytecode verifier? (cont.)

We will define a bytecode verifier for a fragment of JVMML.

- Its definition is a set of typing rules.
- It is sound.
- It is close to Sun's.

Bytecode verification, Part II

Reading

A paper by Stata and Abadi:

- “A type system for Java bytecode subroutines” ,
reachable from www.soe.ucsc.edu/~abadi/allpapers.html.

MicroJVML

MicroJVML is a tiny subset of JVMML.

MicroJVML programs are instruction sequences.

In MicroJVML, there are only 7 instructions.

MicroJVML instructions

$instruction ::= inc$
| `pop`
| `push0`
| `load x`
| `store x`
| `if L`
| `halt`

where x ranges over Var (the set of variables),
and L ranges over Addr (the set of addresses).

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.

Small-step semantics

A small-step semantics uses the judgement:

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

Some rules for this judgement are: ($n \in \text{Int}, L \in \text{Addr}$)

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

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

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

$$\frac{P[pc] = \text{if } L \\ n \neq 0}{P \vdash \langle pc, f, n \cdot s \rangle \rightarrow \langle L, f, s \rangle}$$

The remaining rules (for reference)

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

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

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

A bytecode verifier for MicroJVML

The typing rules of MicroJVML should prevent:

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

but they should allow local typing.

Main typing judgement

$F, S \vdash P$

$F_{pc}[x]$ type of variable x at point pc
(possibly undefined)

S_{pc} type of operand stack at point pc

Top-level typing rule

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

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 .

Typing rule for inc

$$P[i] = \text{inc}$$
$$F_{i+1} = F_i$$
$$S_{i+1} = S_i = \text{Int} \cdot \alpha$$
$$i + 1 \in \text{Dom}(P)$$

$$F, S, i \vdash P$$

Typing rule for load

$P[i] = \text{load } x$

$x \in \text{Dom}(F_i)$

$F_{i+1} = F_i$

$S_{i+1} = F_i[x] \cdot S_i$

$i + 1 \in \text{Dom}(P)$

$F, S, i \vdash P$

Main 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 $s : Spc$

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

In other words, if a program typechecks, then it makes progress until it halts.

\Rightarrow No uncaught errors.

Credits / further work

Raymie Stata and I developed this approach and studied subroutines.

Steve Freund and John Mitchell extended the approach and studied other language features.

Freund's dissertation (2000)

Features covered:

- classes,
- interfaces,
- virtual methods,
- arrays,
- integers and floating point numbers,
- object initialization and constructors,
- subroutines,
- exceptions.

⇒ all complex features of JVM.

Freund's dissertation (2000)

- ~ 100 rules for static analysis.
- ~ 50 rules in machine description.
- 8 major invariants.
- 120 lemmas.
- Over 150 pages of proof.
- Techniques for managing the proof complexity.
- A typechecking algorithm.
- An implementation.

Related work

- Bershad et al. (experimental),
- Cohen (“defensive Java Virtual Machine”),
- Coglio, Goldberg, and Qian (dataflow),
- Saraswat (concurrent constraint programming),
- Hagiya and Tozawa (more subroutines),
- O’Callahan (even more subroutines),
- Yelland, Jones (Haskell view),
- Pusch (mechanical proofs),
- Dean, Saraswat, Bracha and Liang,
Coglio and Goldberg (class loading),
- Rose and Rose,
- Possegga and Vogt,

...

Bytecode verification, Part III

Compiling Java **try-finally**

```
int bar(int i) {  
    try {  
        if (i == 3) return this.foo();  
    } finally {  
        this.ladida();  
    }  
    return i;  
}
```

finally body is compiled into a subroutine.

The subroutine is called from each escape point.

```

01 iload_1           // Push i
02 iconst_3         // Push 3
03 if_icmpne 10     // Goto 10 if i does not equal 3
// Then case of if statement
04 aload_0         // Push this
05 invokevirtual foo // Call this.foo
06 istore_2        // Save result of this.foo()
07 jsr 13          // Execute finally block
// before returning
08 iload_2         // Recall result from this.foo()
09 ireturn         // Return result of this.foo()
// Else case of if statement
10 jsr 13         // Do finally block
// before leaving try
// Return statement following try statement
11 iload_1         // Push i
12 ireturn        // Return i
// finally block
13 astore_3       // Save return address
14 aload_0       // Push this
15 invokevirtual ladida // Call this.ladida()
16 ret 3         // Return from finally block
// Exception handler for try body
17 astore_2     // Save exception
18 jsr 13      // this.foo raised an exception
19 aload_2     // Recall exception
20 athrow     // Rethrow exception
// Exception handler for finally body
21 athrow     // Rethrow exception

```

Typing challenges of subroutines

Polymorphism:

- The callers of a subroutine need not agree on the types of all local variables.
- The callers must agree only on the types of local variables used by the callee.
- Polymorphism is needed in practice.
(Example: return value vs. exception object.)

LIFO behavior:

- Subroutines are not obviously LIFO.
- But LIFO behavior helps type analysis.
(Static analysis with computed goto's is hard.)

Typing rule for jsr

$$\begin{array}{c} P[i] = \text{jsr } L \\ \text{Dom}(F_{i+1}) = \text{Dom}(F_i) \\ \text{Dom}(F_L) \subseteq \text{Dom}(F_i) \\ \forall y \in \text{Dom}(F_i) \setminus \text{Dom}(F_L). F_{i+1}[y] = F_i[y] \\ \forall y \in \text{Dom}(F_L). F_L[y] = F_i[y] \\ S_L = (\text{ret-from } L) \cdot S_i \wedge (\text{ret-from } L) \notin S_i \\ \forall y \in \text{Dom}(F_L). F_L[y] \neq (\text{ret-from } L) \\ i + 1 \in \text{Dom}(P) \\ L \in \text{Dom}(P) \\ \hline F, S, i \vdash P \end{array}$$

where $(\text{ret-from } L)$ is a type of return addresses.

+ Restrictions on call graph forbid recursion.

Typing rule for `ret`

$$P[i] = \text{ret } x$$

i is in routine that starts at L

$$F_i[x] = (\text{ret-from } L)$$

$$\forall j. P[j] = \text{jsr } L \Rightarrow$$

$$\left(\begin{array}{l} \wedge \forall y \in \text{Dom}(F_i). F_{j+1}[y] = F_i[y] \\ \wedge S_{j+1} = S_i \end{array} \right)$$

$$F, S, i \vdash P$$

Compiling Java's object initialization

```
Point p = new Point(3);  
p.print();
```

may become:

```
01 new Point  
02 dup  
03 iconst_3  
04 invokespecial <method Point(int)>  
05 dup  
06 invokevirtual <method void print()>
```

where there are two references to the same uninitialized object.

⇒ We must track aliasing relations.

Goal: no object is used before it is initialized.

Specified implementation:

- For each `new A` instruction, at runtime, there is never more than one uninitialized object created at that instruction.
- For each object, we record type, initialization status, and (if uninitialized) line of origin.
- Two uninitialized objects with the same line of origin are aliases.

A flaw in Sun's JDK1.1.4

Freund found this while studying JVMIL typing:

```
01 jsr 09 // Go allocate a P
02 store 1 // Put it in 1
03 jsr 09 // Go allocate another P
04 store 2 // Put it in 2
    // Two objects with same type and origin!
05 load 2 // Recall 2
06 init P // Initialize 2
07 load 1 // Recall 1
08 use P // Use 1 without initializing it!
    // Subroutine for allocation:
09 store 0 // Store return address
10 new P // Allocate a P
11 ret 0 // Return
```

(To our knowledge, this does not permit actual security violations.)

Doing without `jsr` and `ret`

In hindsight, the use of `jsr` and `ret` is not worth the trouble (in comparison with inlining).

There have been efforts to make them disappear.

Bytecode verification, Part IV

How far does typing go?

Typing provides a base level of security.

- Protection and attacks can be understood in terms of the language semantics.
- Typing problems can suggest security problems.

But typing is not the same as integrity or secrecy.

- A **private** field may be world-writable.

Typing yields only partial bounds on resource use and usually does not guarantee availability.

From languages towards systems

Security is a property of systems.

A language is not a system.

A system may include compilers, servers, users,

The gap between Java and JVMIL is a common source of trouble.

The gap between C# and MSIL is at least as problematic (see Andrew Kennedy).

Full abstraction

The idea of “full abstraction” is useful for understanding the security of implementations

- in the case of Java and C[#],
- for communication abstractions (e.g., RMI),
- perhaps in general.

Loosely:

- Two programs are equivalent in a language if they have the same observable behavior in all contexts of the language (Morris).
- A translation is fully abstract if it respects equivalence (Milner, Plotkin, Mitchell, ...).

A class in Java

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

Conjecture:

In Java, any two instances of `c` are equivalent.

⇒ A Java programmer may expect the value of `x` to remain secret.

The same class in JVMIL

```
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  
    };  
}
```

Conjecture:

In JVMIL, any two instances of `C` are equivalent (thanks to bytecode verification).

An inner class in Java

```
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 with access to D's x.

Translation of an inner class into JVMIL

```
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 ...  
}
```

JVML does not have inner classes \Rightarrow get_x is added to D.

Security and full abstraction

Since verification restricts JVMIL contexts, it “pushes” towards security and full abstraction.

But:

Any class in the same package as `D` can access `x`.

⇒ Despite bytecode verification, the translation from Java to JVMIL does not preserve all “expected” security properties.

A JVMIL context that runs `get_x` distinguishes instances of `D` with different values for `x`.

⇒ Despite bytecode verification, full abstraction is lost.

Security and full abstraction (cont.)

What is a secure implementation of Java?

- Full abstraction could be nice.
- It may be too much to ask for.
- Weaker guarantees may suffice.

Alternatives to full abstraction

One may ignore the security of translations

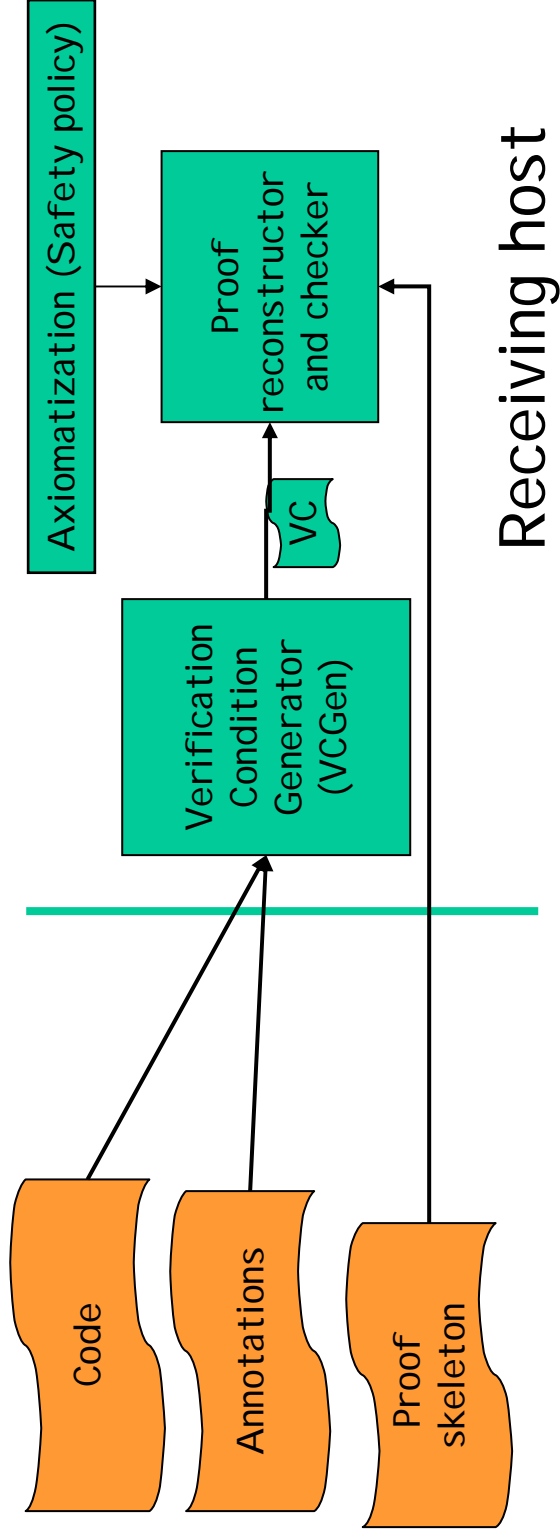
- when low-level code is signed by a trusted party,
- if one analyzes low-level code independently of its source.

These alternatives are not always satisfactory.

In other cases, translations should preserve at least some security properties.

Proof-carrying code (PCC)

A basic PCC architecture



Features

Proofs may be hard, but checking may be simple.

With PCC, the code may be close to what is actually run, e.g., assembly-language code.

A complex compiler need not be trusted.

The checker may work for multiple safety policies, not just memory safety.

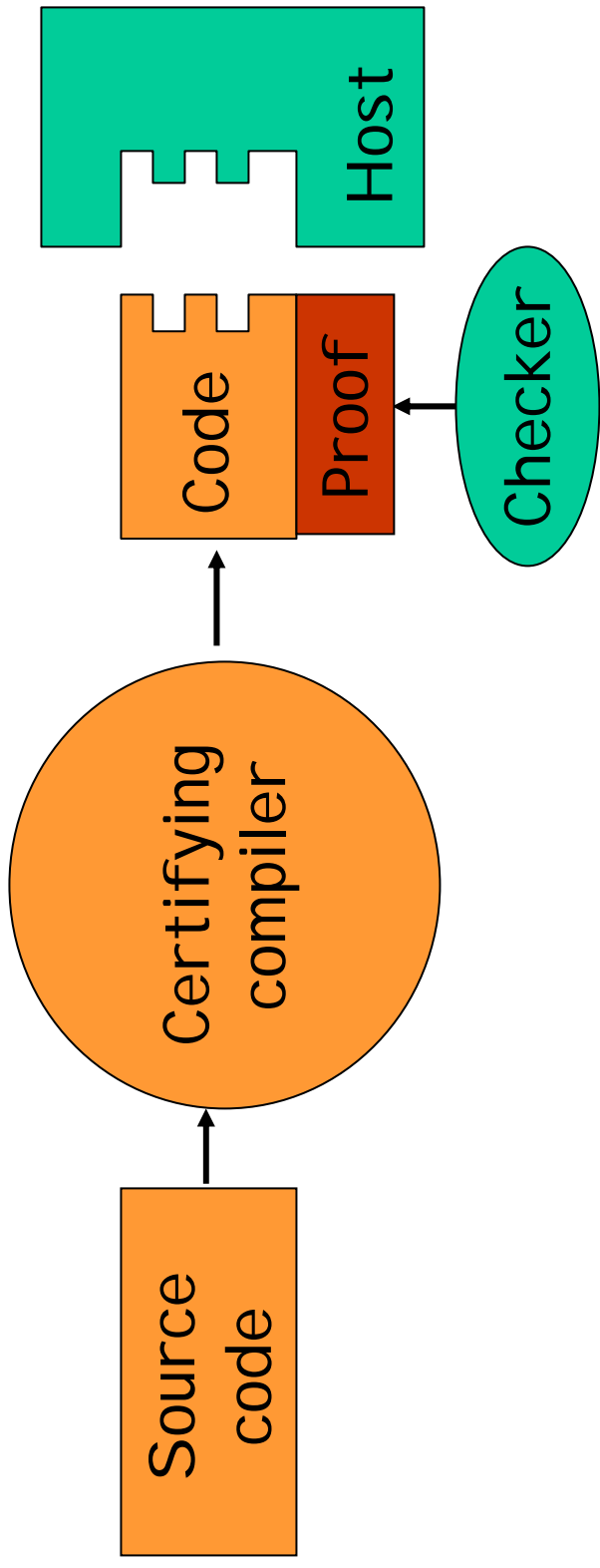
Challenges

- Generating the loop invariants.
- Generating the proofs.

An idea (for type safety):

- Start with a type-safe language.
- Translate source typings into lower-level proofs as part of compilation.

Certifying compilers



(There are such compilers, still as research artifacts.)

Contrast with Java bytecode verification.

Information-flow control

Reading

A paper by Palsberg and Ørbæk:

- “Trust in the λ -calculus”,
reachable from

www.cs.ucla.edu/~palsberg/publications.html.

Information-flow control

Untrusted code should respect the confidentiality and integrity of one's data.

- Data can be classified into levels, or types, like Secret and Public, like Trusted and Untrusted.
- Dynamically, an interpreter may enforce that:
 - secret data does not leak,
 - trusted data does not get corrupted,e.g., don't put Secret data in a Public address.
- For example, versions of Perl and JavaScript have some dynamic checks for confidentiality.

Some complications

- Suppose that s is supposed to remain secret.

```
if  $s$  then broadcast 1
   else broadcast 0
```

This is an implicit flow.

- In information-flow models, security is often a property of sets of behaviors.

```
if  $s$  then broadcast 1
   else broadcast 1
```

⇒ Checking each behavior as it happens can provide only an approximation to security.

Static information-flow control

Information-flow checks can also be static by verification or typechecking (see Denning).

- ⇒ Saving in runtime cost.
- ⇒ Analysis of sets of behaviors.
- ⇒ Early trouble detection.

Static information-flow control

Much recent work develops the static approach.

- Palsberg and Ørbæk,
- Volpano, Smith, and Irvine,
- Heintze and Riecke,
- Myers and Liskov,

:

Some trends:

- better proof techniques,
- nondeterminism, probabilities, concurrency,
- richer type structures,
- living with declassification.

(See me for references!)

A typical rule

Let $\text{Public} \leq \text{Secret}$.

Let $h, k \in \{\text{Secret}, \text{Public}\}$.

$$\begin{array}{l} E \vdash a : (\text{Int}, h) \\ E \vdash b : (\text{Int}, k) \end{array}$$

$$E \vdash a + b : (\text{Int}, \max(h, k))$$

A typical property about information flow

Suppose that program P typechecks, is deterministic, and has variables:

x of type (Int, Secret),

y of type (Int, Public).

Assume that

$x := 0; y := 0; P$

terminates with $y = n$.

Then, for all integers m ,

$x := m; y := 0; P$

also terminates with $y = n$.

So looking at the final value of y does not reveal anything about the initial value of x .

Homework 4 (for May 4)

Exercise 1:

Tail-call elimination is the following optimization: instead of building a new stack frame for the last call in a function, the current frame is overwritten so that the callee directly returns to the caller's caller. Give a small example in which tail-call optimization may create a security problem when stack inspection is used.

Homework 4 (cont.)

Exercise 2:

According to the type system of section 4 of the paper by Stata and Abadi, are the following programs well-typed?

(yes/no)

- a) `inc pop halt`
- b) `halt pop halt`
- c) `push0 pop halt`

(You should rely on the typing rules, not the small-step semantics rules. If you do proofs, you may partly rely on the style of Figure 9.)

Homework 5 (for May 11)

In the type system of Figure 8 of the paper by Palsberg and Ørbæk, are there types for the following expressions? (yes/no)

- a) $\text{check}(\lambda x. x)$
- b) $\text{check}(\lambda x. \text{check } x)$
- c) $\text{check}(\lambda x. \text{distrust}(\text{trust } x))$
- d) $\text{check}(\text{distrust}(\lambda x. x))$