Improving Usability of Information Flow Security in Java

Mark Thober

Joint work with Scott F. Smith

Department of Computer Science

Johns Hopkins University

Motivation

- Information security is a critical requirement of software systems
  - Personal information, trade secrets, national security, etc.

Motivation

- Information security is a critical requirement of software systems
  - Personal information, trade secrets, national security, etc.
- Secure information flow type systems have been around for a long time, but are not in widespread use
  - The burden on programmers is great: many annotations, unclear policies, complex reasoning
  - Overly restrictive type systems reduce the expressiveness and flexibility of programs

Motivation

- Information security is a critical requirement of software systems
  - Personal information, trade secrets, national security, etc.
- Secure information flow type systems have been around for a long time, but are not in widespread use
  - The burden on programmers is great: many annotations, unclear policies, complex reasoning
  - Overly restrictive type systems reduce the expressiveness and flexibility of programs

Goal: Give programmers better tools to write information flow secure programs

Usability [Nielsen '93]

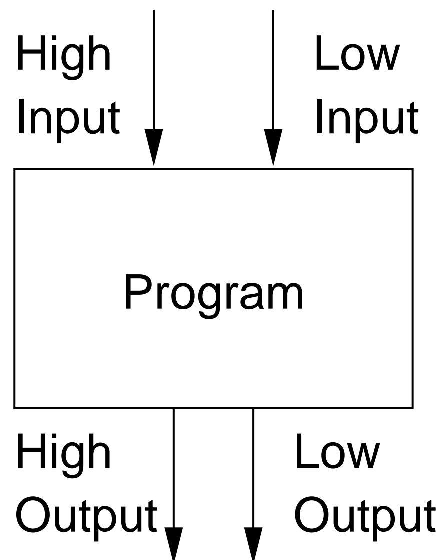
- **Learnability**: users should quickly be able to use the system
- **Efficiency**: once learned, users should be highly productive
- **Memorability**: after non-use, users should be able to return without much relearning
- **Errors**: minimize errors and increase recoverability of errors
- **Satisfaction**: users should (subjectively) like it

Our Approach

- Static information flow type inference for Middleweight Java
- IO points explicitly specify the security policy
- Infer *all* other security labels
  - Program annotations not needed (except for IO points)
  - Less burden on programmers; errors are less likely, alleviating the programmer's responsibility of writing correct annotations
- High degree of polymorphism
  - Increases precision, fewer secure programs rejected
  - Flexible re-use of code across security domains
- Top-level declarations clarify the security policy

IO Focus

- IO points explicitly specify the security policy
  - Internal checks are not necessary



- The type inference system ensures high inputs do not affect low outputs (Noninterference)

Example

```
class HighFileIS extends FileIS
{ int read() { return read_High(fd); } }

class LowFileIS extends FileIS
{ int read() { return read_Low(fd); } }

class LowFileOS extends FileOS
{ void write(int v) { write_Low(v,fd); } }

void main() {
 FileIS highin = new HighFileIS("high_infile");
 FileIS lowin = new LowFileIS("low_infile");
 FileOS lowout = new LowFileOS("low_outfile");
 int x; int y;

 x = lowin.read();
 y = highin.read();
 lowout.write(x);
 lowout.write(y);
}
```

Example

```
class HighFileIS extends FileIS
{ int read() { return read_High(fd); } }

class LowFileIS extends FileIS
{ int read() { return read_Low(fd); } }

class LowFileOS extends FileOS
{ void write(int v) { write_Low(v,fd); } }

void main() {
 FileIS highin = new HighFileIS("high_infile");
 FileIS lowin = new LowFileIS("low_infile");
 FileOS lowout = new LowFileOS("low_outfile");
 int x; int y;

 x = lowin.read();
 y = highin.read();
 lowout.write(x); // Passes
 lowout.write(y); // Fails
}
```

Observe

- Only low-level IO declarations change

```
class HighFileIS extends FileIS
{
 int read() { return read_High(fd); }
}
```

  - Signature of the class, and accessor methods do *not* change
- Everything else is inferred
  - Apart from the IO declarations, programs are in Java
  - No additional internal annotations are needed
- Programmers must use separate IO classes in order to distinguish the security policies

Polymorphism

Two purposes:

- Distinguish Input and Output classes, which may have different security policies
- Permit re-use of code across security domains
  - Should not be overly conservative

Example Streams, again

```
class HighFileIS extends FileIS
{
 int read() { return read_High(fd); }
}
class LowFileIS extends FileIS
{
 int read() { return read_Low(fd); }
}
class LowFileOS extends FileOS
{
 void write(int v) { write_Low(v, fd); }
}
```

Polymorphism Example

```
void main() {
 int i; int j;

 HashSet highSet = new HashSet();
 FileIS hin = new HighFileIS("high_infile");
 while(i = hin.read()) { highSet.add(i); }

 HashSet lowSet = new HashSet();
 FileIS lin = new LowFileIS("low_infile");
 while(j = lin.read()) { lowSet.add(j); }

 Iterator lowIt = lowSet.iterator();
 FileOS lowout = new LowFileOS("low_outfile");
 lowout.write(lowIt.next());
}
```

Polymorphism Example

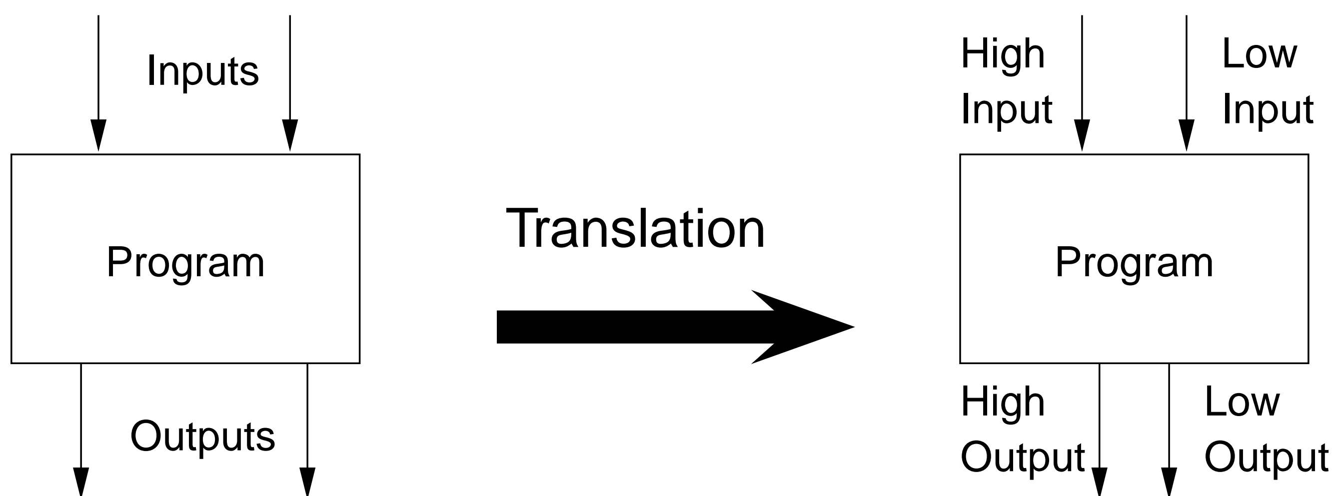
```
void main() {
 int i; int j;

 HashSet highSet = new HashSet();
 FileIS hin = new HighFileIS("high_infile");
 while(i = hin.read()) { highSet.add(i); }

 HashSet lowSet = new HashSet();
 FileIS lin = new LowFileIS("low_infile");
 while(j = lin.read()) { lowSet.add(i); }

 Iterator lowIt = lowSet.iterator();
 FileOS lowout = new LowFileOS("low_outfile");
 lowout.write(lowIt.next()); // No leakage!
}
```

Top-level Policies

- Clarify the policy in the API

Top-level Policies

- Clarify the policy in the API
- Add security policies to an existing program
  - The program internals don't change, only the policy



Top-level Policies

- Read and write policies add security labels to methods of input and output stream classes
- Declassify policy adds a declassification to the return value of a method

Top-level Policies

- Read and write policies add security labels to methods of input and output stream classes
- Declassify policy adds a declassification to the return value of a method

Example Policies

```
class HighFileIS: High
class LowFileIS: Low
class LowFileOS: Low
class TripleDES
 byte[] Encrypt (byte[] input, SecretKey key):
  Declassify(Low)
```

Top-level Policies

- Channel policies are evident in API
- Restricting declassification to method returns clarifies the declassification policy and makes it observable in the API
- Observe top-level policies to decide if declassification is intuitively warranted
  - Some knowledge of the underlying code may be required to certify declassifications

Assumptions

- Direct and indirect flows
  - Direct:

```
x = h + 1;
```

  - Indirect:

```
if (h == 0) then {x = 0;} else { }
```

- No termination, timing, or other covert channels
- Declassification is allowed, but breaks Noninterference

Language

- Extension of Middleweight Java (MJ)
  - Core object-oriented features of Java, including state
- Add constants (`int`, `bool`, etc.) and operators (+, −, etc.)
- Add low-level reads and writes $\mathtt{read}_{\mathrm{L}}(\mathtt{fd})$ and $\mathtt{write}_{\mathrm{L}}(\mathtt{e}, \mathtt{fd})$
  - `fd` is the file descriptor naming the channel
  - `L` is the security level of the channel
- Add `Declassify(e,L)`, which downgrades expression e

Language

- Extension of Middleweight Java (MJ)
  - Core object-oriented features of Java, including state
- Add constants (`int`, `bool`, etc.) and operators (+, −, etc.)
- Add low-level reads and writes $\texttt{read}_{\mathrm{L}}(\texttt{fd})$ and $\texttt{write}_{\mathrm{L}}(\texttt{e}, \texttt{fd})$
  - `fd` is the file descriptor naming the channel
  - `L` is the security level of the channel
- Add `Declassify(e,L)`, which downgrades expression `e`
- End goal is full Java
  - A core subset with full formalism and proofs is a first step

Type Inference

- Types τ are triples, $\langle \mathcal{S}, \mathcal{F}, \mathcal{A} \rangle$
  - $\mathcal{S}$ are security types that may be concrete labels, $\mathtt{L}$, or label variables, when the concrete label is unknown
  - $\mathcal{F}$ are the types of the object's fields
  - $\mathcal{A}$ is a concrete class type for statically determining the concrete class of an object
- Type constraints encapsulate certain information flows, and the constraint set must be closed after type inference

Class and Program Typing

- Requires a *Label Table* that contains the type of each class and method

1. Initialize a Label Table with type variables
   - Must initialize all classes to allow for recursion
2. Propagate the label table by typing each class, which requires typing constructors and method bodies
3. Type `main`
4. Compute the constraint closure and check for inconsistencies

Typing Reads

The security type of $\mathtt{read_L(e)}$ is $\mathtt{L}$, the security level of the channel

- A check is also performed to eschew information leaks
  - Low reads under high guards can leak information

    ```
    if (h > 0) { read_Low(fd); }
    ```

    Potential mismatch in the low stream `fd` in different runs
  - The file descriptor can also leak information

    ```
    if (h > 0) then {x = fd;} else {x = fd';}
    read_Low(x);
    ```

    The low observer can determine if `h > 0` based on whether `fd` or `fd'` is read

Typing Writes

Typing $\texttt{write}_{\texttt{L}}(\texttt{e}, \texttt{e}')$ invokes a similar security check

- Data being written, $\texttt{e}$, should conform to the policy of the channel, $\texttt{L}$

  $\texttt{write}_{\texttt{Low}}(\texttt{x}, \texttt{fd});$

  This will only type check if $\texttt{x}$ is directly and indirectly low

- As in typing reads, all indirect flows must conform to the type of the channel; assuming $\texttt{h}$ is high data, the following examples both leak information (and are therefore *not* typeable)

  $\texttt{if (h > 0) \{ write}_{\texttt{Low}}(\texttt{5}, \texttt{fd}); \texttt{ \}}$

  $\texttt{if (h > 0) then \{x = fd;\} else \{x = fd}';\}$
  $\texttt{write}_{\texttt{Low}}(\texttt{5}, \texttt{x});$

Security Checks

Security checks are a special constraint, $SC(\mathtt{L}, \mathcal{S})$, where $\mathtt{L}$ is the security level of the channel, and $\mathcal{S}$ is the type to be checked

- Necessary since some labels are not immediately known

```
public Class LowFileOS extends FileOS {
 FileDescriptor fd;
 public int write (int x)
  { write_Low(x, fd); }
}
```

- At constraint closure, the type of $\mathcal{S}$ is made concrete as some $\mathtt{L}'$, such that $\mathtt{L}' \leq \mathtt{L}$ is consistent
  - $SC(\mathtt{High}, \mathtt{Low})$ is consistent
  - $SC(\mathtt{Medium}, \mathtt{High})$ is inconsistent

Method Typing

- Need a more precise analysis to achieve better polymorphism
  - An object of type `FileIS` may at run-time be a subclass, and different subclasses of `FileIS` may have different policies
- Concrete class analysis: statically determine which concrete classes an object can be
  - CPA-style [Agesen '95, *etc.*] with let-polymorphism
- Method bodies are universally quantified with a $\forall$ type
- Different method calls create unique *contours* (polyinstantiations) of the $\forall$ type, giving the increased polymorphic expressiveness we require

See the paper for details

Example Typing

```
class HighFileIS extends FileIS
 { int read() { return read_High(fd); } }
class LowFileIS extends FileIS
 { int read() { return read_Low(fd); } }
class LowFileOS extends FileOS
 { void write(int v) { write_Low(v, fd); } }
class C extends Object
 { int x; }
...
FileIS lin;
FileIS hin;
...
lowC = new C(0);
highC = new C(0);
lowC.x = lin.read();
highC.x = hin.read();
lowout.write(lowC.x);
...
```

Example Typing

```
class HighFileIS extends FileIS
 { int read() { return read_High(fd); } }
class LowFileIS extends FileIS
 { int read() { return read_Low(fd); } }
class LowFileOS extends FileOS
 { void write(int v) { write_Low(v, fd); } }
class C extends Object
 { int x; }
...
FileIS lin;
FileIS hin;
...
lowC = new C(0);
highC = new C(0);
lowC.x = lin.read();
highC.x = hin.read();
lowout.write(lowC.x);
...
```

- The type system must distinguish object instances and method calls

Example Typing, (2)

$\texttt{class HighFileIS extends FileIS}$
$\{\ \texttt{int read()}\ \{\ \texttt{return read}_{\texttt{High}}(\texttt{fd});\ \}\ \}$

- Generates a constraint showing the return type of the method is $\texttt{High}$

$\texttt{class LowFileIS extends FileIS}$
$\{\ \texttt{int read()}\ \{\ \texttt{return read}_{\texttt{Low}}(\texttt{fd});\ \}\ \}$

- Generates a constraint showing the return type of the method is $\texttt{Low}$

$\texttt{class LowFileOS extends FileOS}$
$\{\ \texttt{void write(int v)}\ \{\ \texttt{write}_{\texttt{Low}}(\texttt{v}, \texttt{fd});\ \}\ \}$

- Generates a constraint $SC(\texttt{Low}, s_v)$, where s_v is the (symbolic) label type of the method argument

Example Typing, (3)

```
lowC = new C(0);
highC = new C(0);
```

- Each time `new` is typed, fresh type variables are created for each object field
  - `lowC.x` and `highC.x` get different types

```
lowC.x = lin.read();
highC.x = hin.read();
```

- Each method invocation gets a unique return type
  - `lin.read()` and `hin.read()` get different types

Example Typing, (4)

$\texttt{lowC} = \texttt{new C}(0);$
$\texttt{highC} = \texttt{new C}(0);$

- $\texttt{lowC.x}$ and $\texttt{highC.x}$ have different types, s_l and s_h

$\texttt{lowC.x} = \texttt{lin.read}();$
$\texttt{highC.x} = \texttt{hin.read}();$

- `lin.read()` and `hin.read()` have different types

$\texttt{lowout.write}(\texttt{lowC.x});$

- So $\texttt{High} <: s_h$, and $\texttt{Low} <: s_l$
- The security check becomes $SC(\texttt{Low}, \texttt{Low})$, which is consistent. **Typing Succeeds!**

Noninterference

If a progam P is well-typed, and two runs of the program with identical low input streams both terminate, then the resulting low output streams are identical (as are the ending low input streams).

Related Work

- Jif [Myers *et. al.* '99, *etc.*]
  - Implemented secure info. flow for full Java. Requires many annotations, no formal noninterference proof
- Secure information flow in a Java-like language [Banerjee and Naumann '02, *etc.*]
  - Formal noninterference proof. Modular inference, yet also requires many parameters to be annotated
- Java Info. Flow based on PDGs [Hammer *et. al.* '06]
  - Significanlty different approach. Similar expressiveness, more complicated formalism, needs further inspection
- Many others, see [Sabelfeld and Myers '03] for a survey

Conclusion

- Static information flow type inference for Middleweight Java
  - Formal correctness proof
- High level of polymorphism promotes IO-based policies and code re-use across security domains
  - Greatly reduces the programmer's burden to annotate
- Top-level policies clarify the security protections

Thanks!

www.cs.jhu.edu/∼mthober