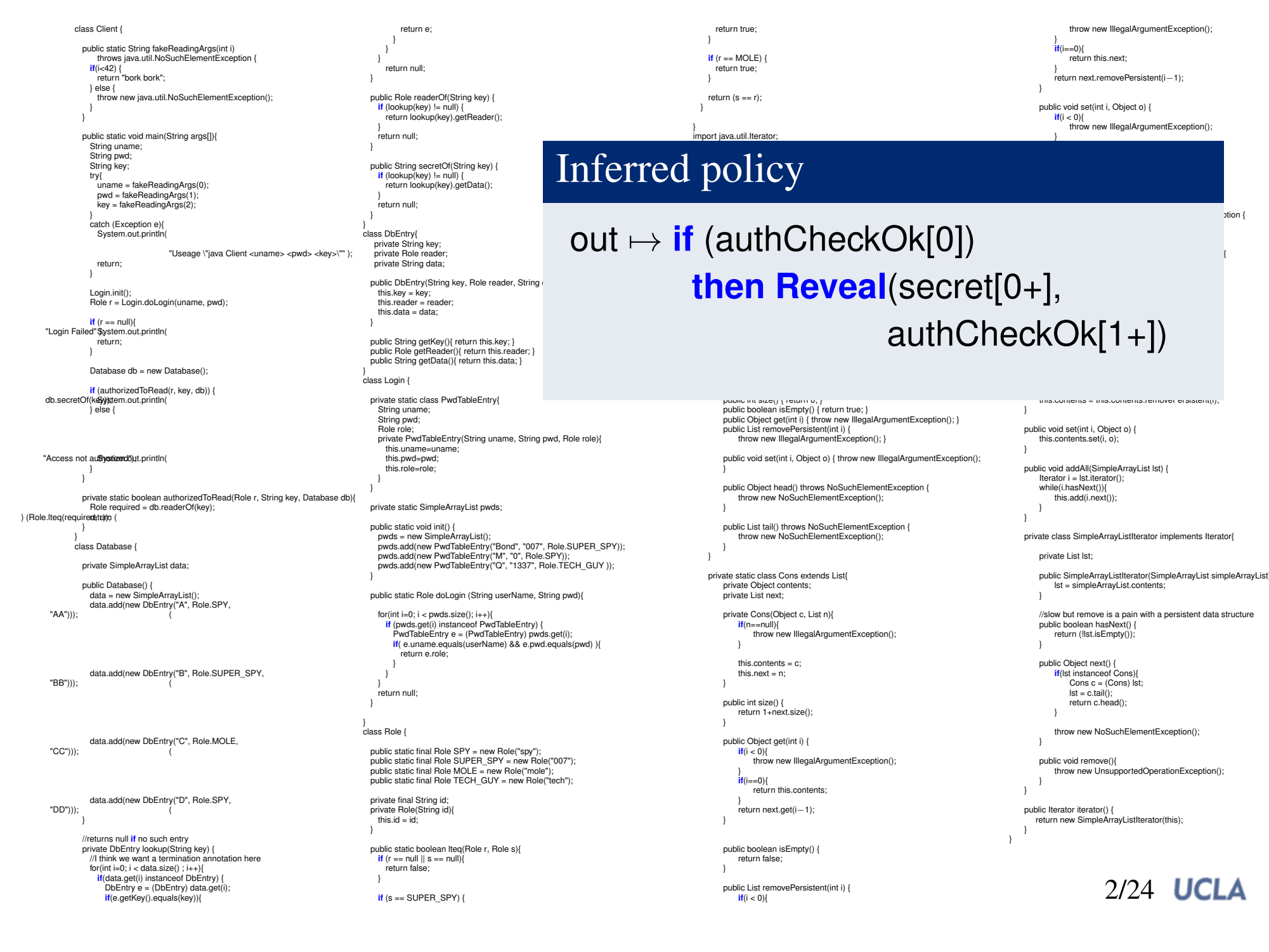
Inference of Expressive Declassification Policies

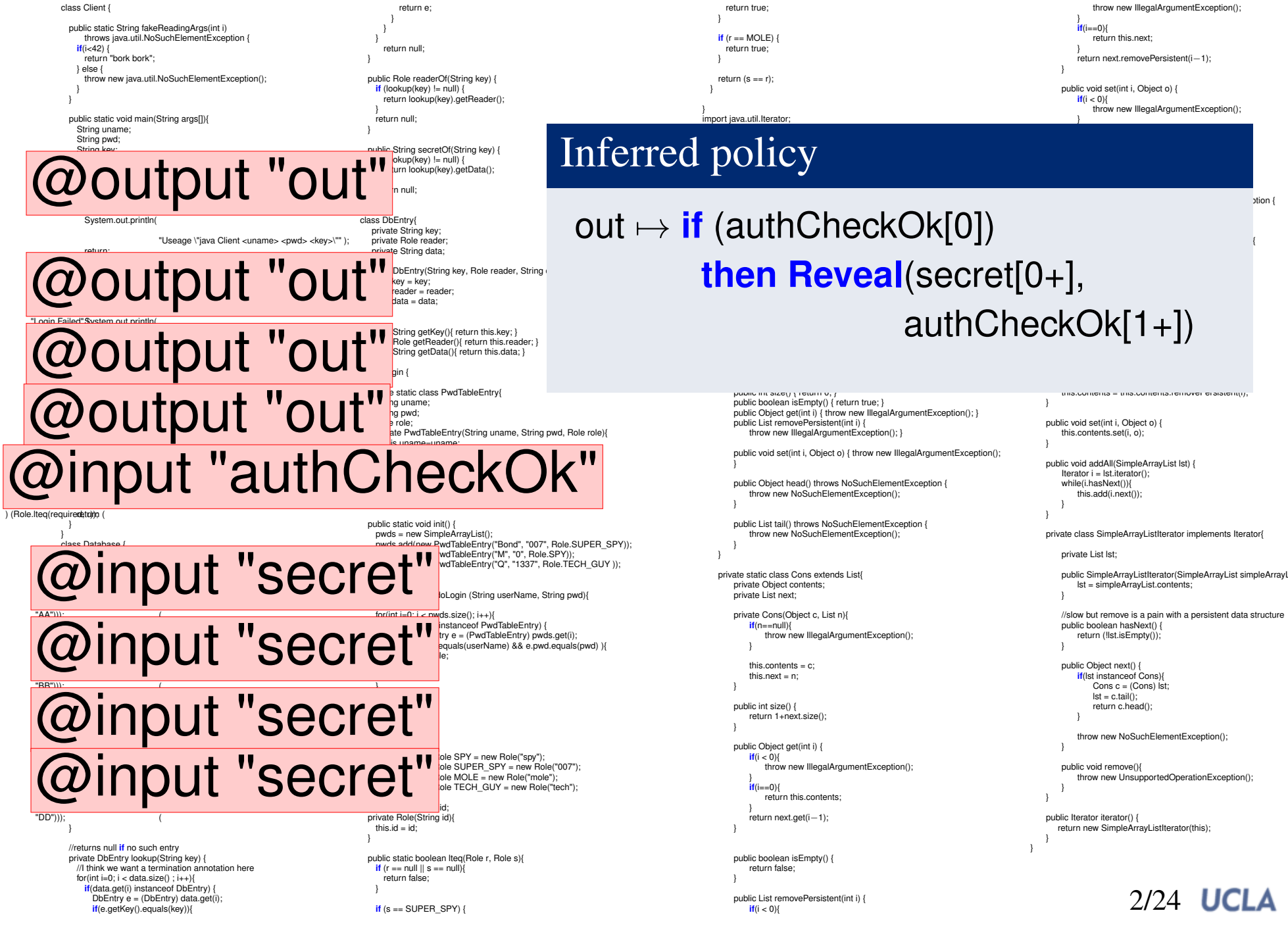
Jeff Vaughan Stephen Chong

UCLA

IEEE Security and Privacy
May 23, 2011

```
class Client {

  public static String fakeReadingArgs(int i)
      throws java.util.NoSuchElementException {
    if(i<42) {
      return "bork bork";
    } else {
      throw new java.util.NoSuchElementException();
    }
  }

  public static void main(String args[]){
    String uname;
    String pwd;
    String key;
    try{
      uname = fakeReadingArgs(0);
      pwd = fakeReadingArgs(1);
      key = fakeReadingArgs(2);
    }
    catch (Exception e){
      System.out.println(

                    "Useage \"java Client <uname> <pwd> <key>\"" );
      return;
    }

    Login.init();
    Role r = Login.doLogin(uname, pwd);

    if (r == null){
"Login Failed");  System.out.println(
      return;
    }

    Database db = new Database();

    if (authorizedToRead(r, key, db)) {
db.secretOf(key));  System.out.println(
    } else {




"Access not authorized");  System.out.println(
    }
  }

  private static boolean authorizedToRead(Role r, String key, Database db){
    Role required = db.readerOf(key);
) (Role.lteq(required, r));  return (
    }
  }
  class Database {

    private SimpleArrayList data;

    public Database() {
      data = new SimpleArrayList();
      data.add(new DbEntry("A", Role.SPY,
"AA")));                    (




      data.add(new DbEntry("B", Role.SUPER_SPY,
"BB")));                    (




      data.add(new DbEntry("C", Role.MOLE,
"CC")));                    (




      data.add(new DbEntry("D", Role.SPY,
"DD")));                    (
    }

    //returns null if no such entry
    private DbEntry lookup(String key) {
      //I think we want a termination annotation here
      for(int i=0; i < data.size() ; i++){
        if(data.get(i) instanceof DbEntry) {
          DbEntry e = (DbEntry) data.get(i);
          if(e.getKey().equals(key)){
            return e;
          }
        }
      }
      return null;
    }

    public Role readerOf(String key) {
      if (lookup(key) != null) {
        return lookup(key).getReader();
      }
      return null;
    }

    public String secretOf(String key) {
      if (lookup(key) != null) {
        return lookup(key).getData();
      }
      return null;
    }
  }
  class DbEntry{
    private String key;
    private Role reader;
    private String data;

    public DbEntry(String key, Role reader, String data){
      this.key = key;
      this.reader = reader;
      this.data = data;
    }

    public String getKey(){ return this.key; }
    public Role getReader(){ return this.reader; }
    public String getData(){ return this.data; }
  }
  class Login {

    private static class PwdTableEntry{
      String uname;
      String pwd;
      Role role;
      private PwdTableEntry(String uname, String pwd, Role role){
        this.uname=uname;
        this.pwd=pwd;
        this.role=role;
      }
    }

    private static SimpleArrayList pwds;

    public static void init() {
      pwds = new SimpleArrayList();
      pwds.add(new PwdTableEntry("Bond", "007", Role.SUPER_SPY));
      pwds.add(new PwdTableEntry("M", "0", Role.SPY));
      pwds.add(new PwdTableEntry("Q", "1337", Role.TECH_GUY ));
    }

    public static Role doLogin (String userName, String pwd){

      for(int i=0; i < pwds.size(); i++){
        if (pwds.get(i) instanceof PwdTableEntry) {
          PwdTableEntry e = (PwdTableEntry) pwds.get(i);
          if( e.uname.equals(userName) && e.pwd.equals(pwd) ){
            return e.role;
          }
        }
      }
      return null;
    }

  }
  class Role {

    public static final Role SPY = new Role("spy");
    public static final Role SUPER_SPY = new Role("007");
    public static final Role MOLE = new Role("mole");
    public static final Role TECH_GUY = new Role("tech");

    private final String id;
    private Role(String id){
      this.id = id;
    }

    public static boolean lteq(Role r, Role s){
      if (r == null || s == null){
        return false;
      }

      if (s == SUPER_SPY) {
        return true;
      }

      if (r == MOLE) {
        return true;
      }

      return (s == r);
    }

  }
  import java.util.Iterator;
  import java.util.NoSuchElementException;

  public class SimpleArrayList implements Iterable {

    List contents;

    SimpleArrayList() {this.contents = new Nil();}

    int size() { return contents.size(); }
    boolean isEmpty() {return contents.isEmpty(); }
    void clear() { this.contents = new Nil(); }

    private static abstract class List {
      private List() { }
      abstract int size();
      abstract boolean isEmpty();
      public abstract Object get(int i);
      public abstract List removePersistent(int i);
      public abstract void set(int i, Object o);
      public abstract Object head() throws NoSuchElementException;
      public abstract List tail() throws NoSuchElementException;
    }


    private static class Nil extends List {
      private Nil(){};
      public int size() { return 0; }
      public boolean isEmpty() { return true; }
      public Object get(int i) { throw new IllegalArgumentException(); }
      public List removePersistent(int i) {
        throw new IllegalArgumentException(); }

      public void set(int i, Object o) { throw new IllegalArgumentException();
      }

      public Object head() throws NoSuchElementException {
        throw new NoSuchElementException();
      }

      public List tail() throws NoSuchElementException {
        throw new NoSuchElementException();
      }
    }

    private static class Cons extends List{
      private Object contents;
      private List next;

      private Cons(Object c, List n){
        if(n==null){
          throw new IllegalArgumentException();
        }

        this.contents = c;
        this.next = n;
      }

      public int size() {
        return 1+next.size();
      }

      public Object get(int i) {
        if(i < 0){
          throw new IllegalArgumentException();
        }
        if(i==0){
          return this.contents;
        }
        return next.get(i−1);
      }


      public boolean isEmpty() {
        return false;
      }

      public List removePersistent(int i) {
        if(i < 0){
          throw new IllegalArgumentException();
        }
        if(i==0){
          return this.next;
        }
        return next.removePersistent(i−1);
      }

      public void set(int i, Object o) {
        if(i < 0){
          throw new IllegalArgumentException();
        }
        if(i==0){
          this.contents = o;
        } else {
          this.next.set(i−1, o);
        }
      }

      public Object head() throws NoSuchElementException {
        return this.contents;
      }

      public List tail() throws NoSuchElementException {
        return this.next;
      }
    }

    //Zero−based lookup
    public Object get(int i) {
      return this.contents.get(i);
    }

    public void add(Object card) {
      this.contents = new Cons(card, this.contents);
    }

    public void remove(int i) {
      this.contents = this.contents.removePersistent(i);
    }

    public void set(int i, Object o) {
      this.contents.set(i, o);
    }

    public void addAll(SimpleArrayList lst) {
      Iterator i = lst.iterator();
      while(i.hasNext()){
        this.add(i.next());
      }
    }

    private class SimpleArrayListIterator implements Iterator{

      private List lst;

      public SimpleArrayListIterator(SimpleArrayList simpleArrayList
        lst = simpleArrayList.contents;
      }

      //slow but remove is a pain with a persistent data structure
      public boolean hasNext() {
        return (!lst.isEmpty());
      }

      public Object next() {
        if(lst instanceof Cons){
          Cons c = (Cons) lst;
          lst = c.tail();
          return c.head();
        }

        throw new NoSuchElementException();
      }

      public void remove(){
        throw new UnsupportedOperationException();
      }
    }

    public Iterator iterator() {
      return new SimpleArrayListIterator(this);
    }
  }
```

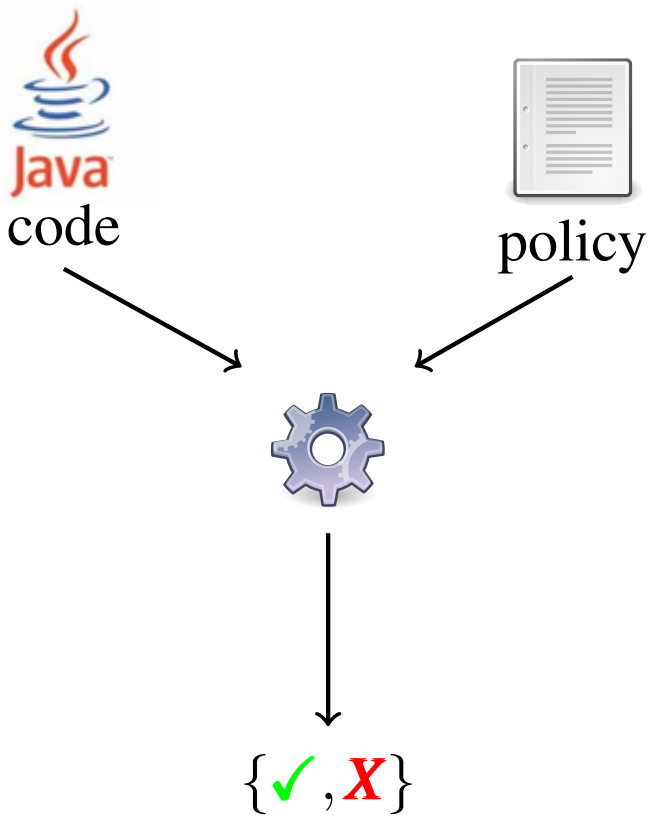

Inferred policy
out ↦ if (authCheckOk[0])
then Reveal(secret[0+],
authCheckOk[1+])



Inferred policy

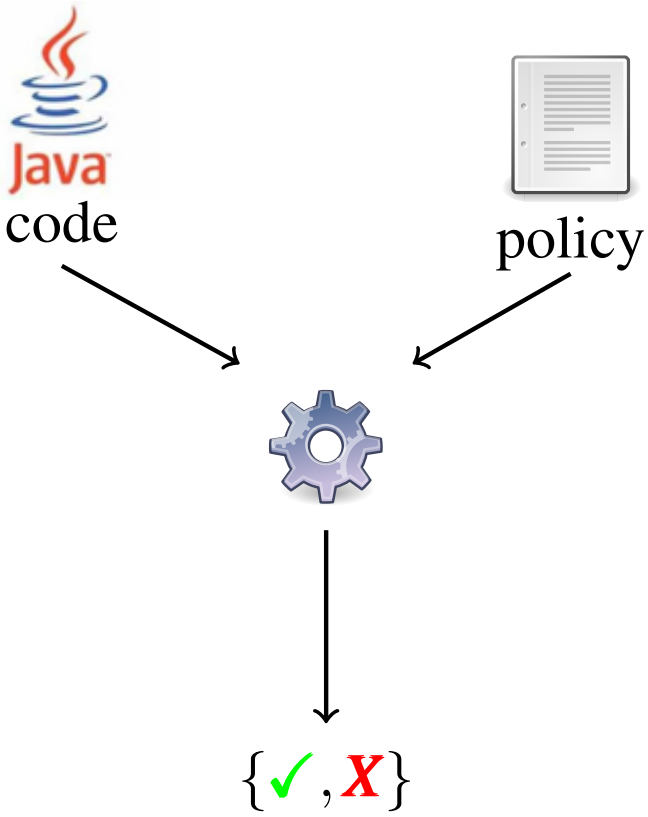
out ↦ **if** (authCheckOk[0])
then Reveal(secret[0+],
authCheckOk[1+])

Goal: Workflow of iterative policy refinement via inference

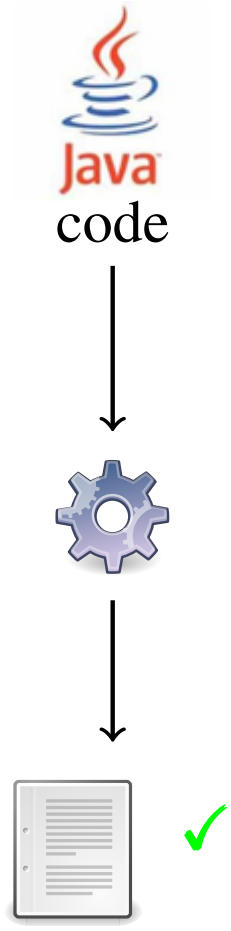


JIF [Myers et. al. '03],
FlowCaml [Simonet '03], ...

Goal: Workflow of iterative policy refinement via inference



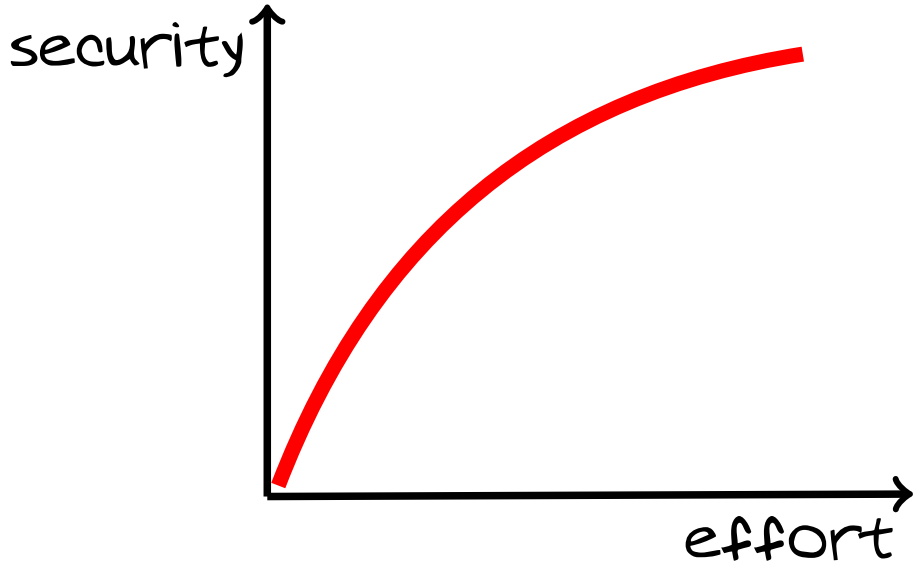
JIF [Myers et. al. '03],
FlowCaml [Simonet '03], . . .



This work

Goal: Analysis suitable for mostly unmodified, legacy Java programs

- Low annotation burden
- Deep analysis of practical Java features: exceptions, heap allocated objects, etc.
- Security guarantees scale with effort



The central tension:

Outline

Reveal policies describe *what* information is released

```
x =                    readln();
y = x;
println(                       y);
```

Reveal policies describe *what* information is released

```
x = @input "stdin" readln();
y = x;
println(@output "stdout" y);
```

Reveal policies describe *what* information is released

```
x = @input "stdin" readln();
y = x;
println(@output "stdout" y);
```

Inferred policy

stdout $\mapsto$ **Reveal**(stdin[0])

Precise input expressions name inputs and derived data exactly

```
String read(){ return                    readln(); }

x = read();
y = read();
println(                          (x == null));
```

Precise input expressions name inputs and derived data exactly

```
String read(){ return @input "H" readln(); }

x = read();
y = read();
println(@output "stdout"  (x == null));
```

Precise input expressions name inputs and derived data exactly

```
String read(){ return @input "H" readln(); }

x = read();
y = read();
println(@output "stdout"  (x == null));
```

Inferred policy

stdout $\mapsto$ **Reveal**(H[1] == null)

H[0] — the most recent H input
H[1] — the second most recent H input
$\vdots$

Imprecise expressions name multiple inputs sharing a label

```
int i = 0;
while (i <= 100)
   i += parse(@input "in" readln());
}
println(@output "out" i);
```

Imprecise expressions name multiple inputs sharing a label

```
int i = 0;
while (i <= 100)
   i += parse(@input "in" readln());
}
println(@output "out" i);
```

Inferred policy

out $\mapsto$ **Reveal**(in[0+])

in[0+] — information from any in input
in[1+] — info. from the second most recent and older in inputs
⋮

Conditional policies describe *when* information is released

```
secret = @input "secret" ...
password = @input "password" ...
guess = @input "guess" ...

if (guess == password) {
   println(@output "stdout" secret);
}
```

Policy?

stdout $\mapsto$ **Reveal**(secret[0], guess[0], password[0])

Conditional policies describe *when* information is released

```
secret = @input "secret" ...
password = @input "password" ...
guess = @input "guess" ...

if (guess == password) {
   println(@output "stdout" secret);
}
```

Inferred policy

stdout $\mapsto$ **if** (guess[0] == password[0]) **then**
Reveal(secret[0])

Track policies describe *where* information is released

```
String safe_release(String x)          {
   if (/* auth check */) {
      return x;
   } else {
      return "";
   }
}
x = @input "in" readln();
y = safe_release(x);
println(@output "out" y);
```

Track policies describe *where* information is released

```
String safe_release(String x) @track {
   if (/* auth check */) {
      return x;
   } else {
      return "";
   }
}
x = @input "in" readln();
y = safe_release(x);
println(@output "out" y);
```

Track policies describe *where* information is released

```
String safe_release(String x) @track {
  if (/* auth check */) {
    return x;
  } else {
    return "";
  }
}
x = @input "in" readln();
y = safe_release(x);
println(@output "out" y);
```

Inferred policy

out $\mapsto$ **if-executed** safe_release(String) **then**
Reveal(in[0])

Inference

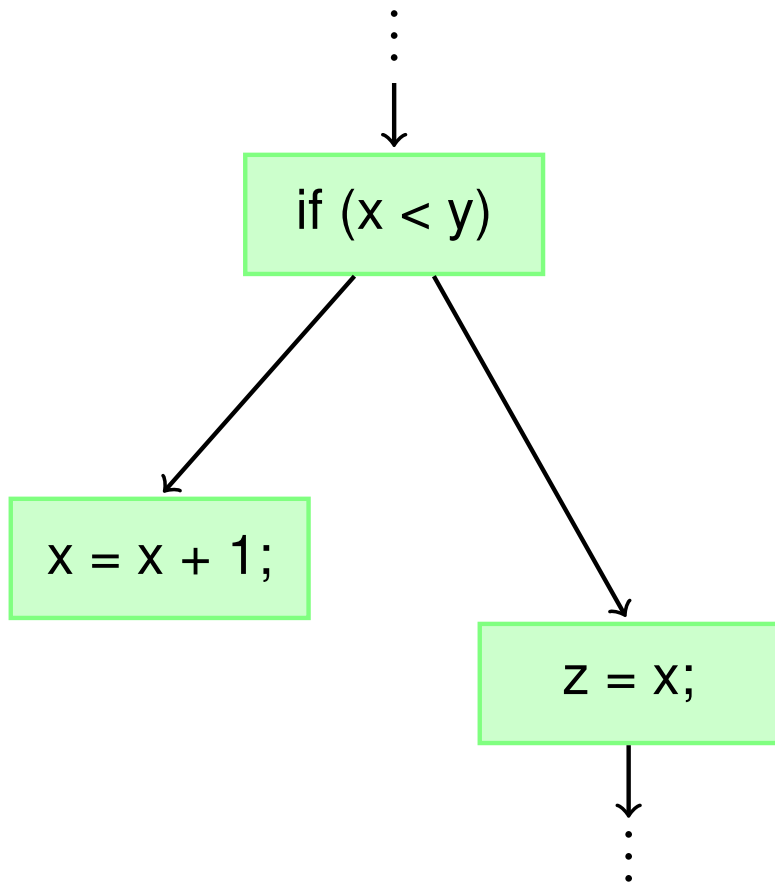
Inference algorithm is based on dataflow analysis

```
if (x < y){
   x = x + 1;
}
z = x;
```

Inference algorithm is based on dataflow analysis

```
if (x < y){
   x = x + 1;
}
z = x;
```

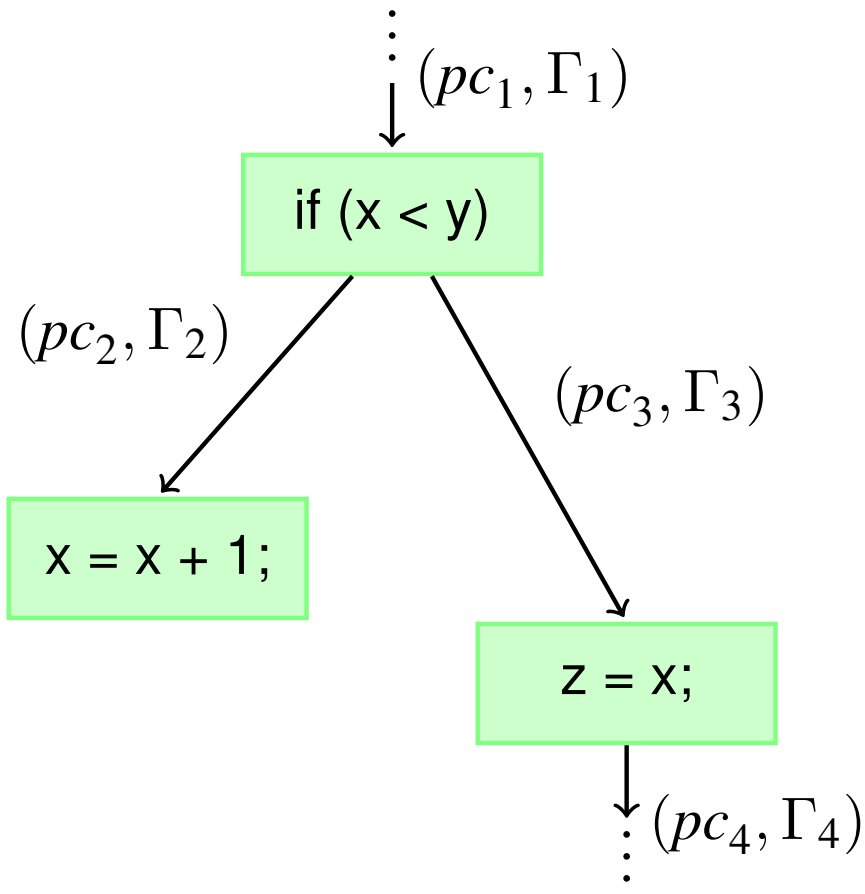
$\Rightarrow$



Inference algorithm is based on dataflow analysis

```
if (x < y){
   x = x + 1;
}
z = x;
```

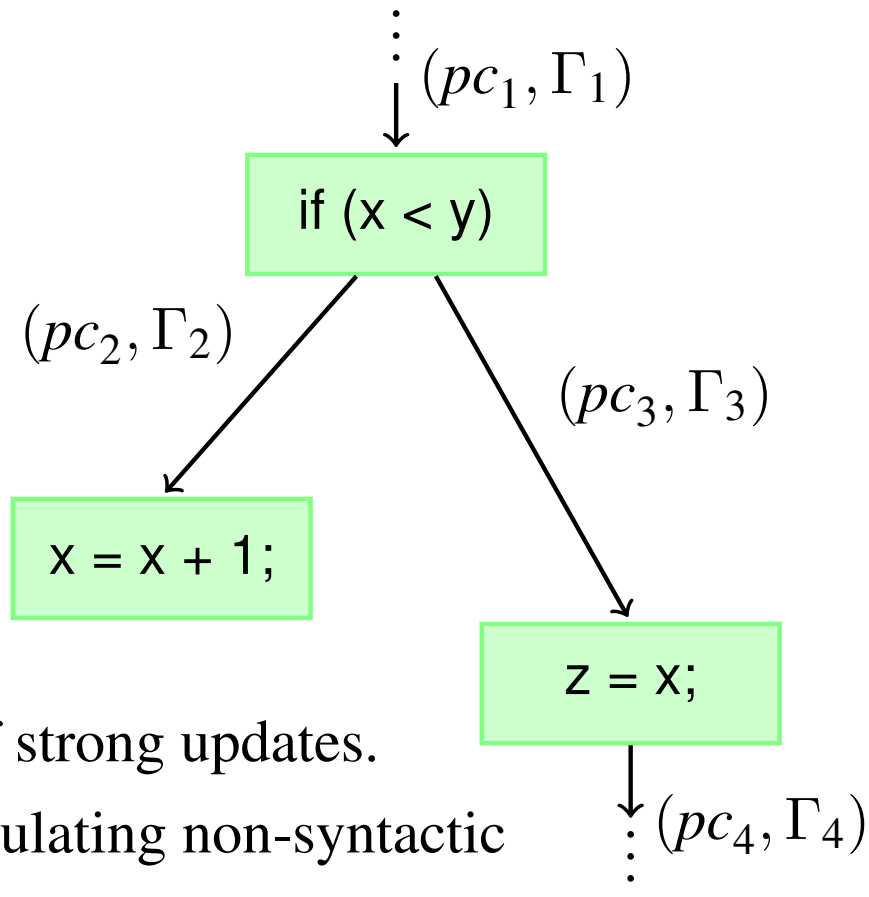
$\Rightarrow$



Inference algorithm is based on dataflow analysis

```
if (x < y){
   x = x + 1;
}
z = x;
```

$\Rightarrow$



- ✓ Natural handling of strong updates.
- ✓ Convenient for calculating non-syntactic control dependencies.

Contexts (pc, Γ) approximates information flow

Location context: information content of fields or heap locations.

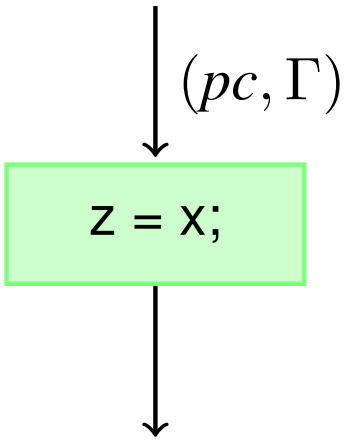
$$\Gamma : \textbf{Location} \rightharpoonup \textbf{Policy}$$

Program counter: what each branch taken to a program point reveals.

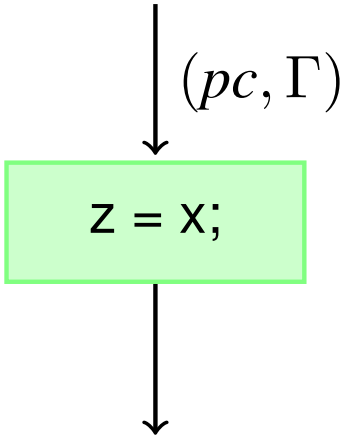
$$pc : \textbf{BranchId} \rightharpoonup \textbf{Policy} \cup \textbf{PIE}$$

Location = abstract vars and refs	**PIE** = precise input exps.
BranchId = static branch ids	**Policy** = policies

Example: Updating a context for an assignment node



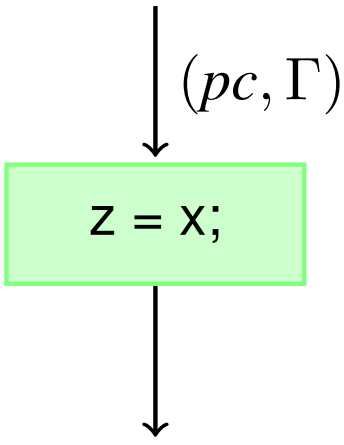
Example: Updating a context for an assignment node



1. Calculate explicit flows.

$$explicit = \Gamma(x)$$

Example: Updating a context for an assignment node



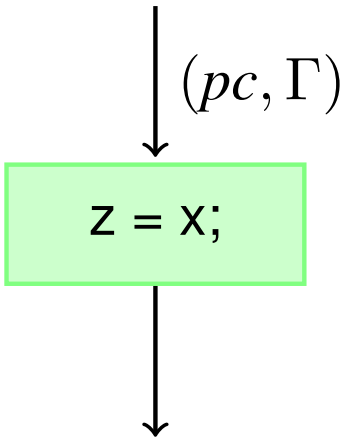
1. Calculate explicit flows.

$$explicit = \Gamma(x)$$

2. Calculate implicit, imprecise flows.

$$implicit = \bigsqcup(range(pc) \cap \textbf{Policy})$$

Example: Updating a context for an assignment node



1. Calculate explicit flows.

$$explicit = \Gamma(x)$$

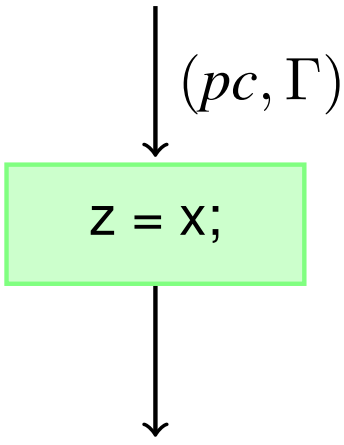
2. Calculate implicit, imprecise flows.

$$implicit = \bigsqcup(range(pc) \cap \textbf{Policy})$$

3. Calculate dominating precise input exps.

$$\{d_1, d_2, \ldots, d_n\} = range(pc) \cap \textbf{PIE}$$

Example: Updating a context for an assignment node



1. Calculate explicit flows.

$$explicit = \Gamma(x)$$

2. Calculate implicit, imprecise flows.

$$implicit = \bigsqcup(range(pc) \cap \textbf{Policy})$$

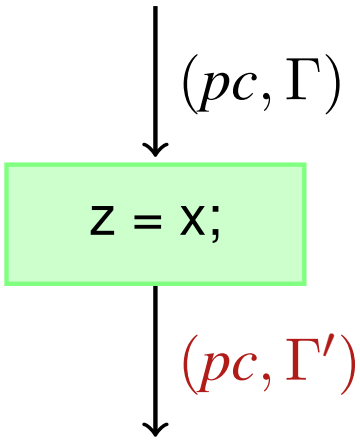
3. Calculate dominating precise input exps.

$$\{d_1, d_2, \ldots, d_n\} = range(pc) \cap \textbf{PIE}$$

4. Assemble a policy.

$pol =$ **if** d_1&& d_2 && .... d_n **then** $(explicit \sqcup implicit)$

Example: Updating a context for an assignment node



1. Calculate explicit flows.

$$explicit = \Gamma(x)$$

2. Calculate implicit, imprecise flows.

$$implicit = \bigsqcup(range(pc) \cap \textbf{Policy})$$

3. Calculate dominating precise input exps.

$$\{d_1, d_2, \ldots, d_n\} = range(pc) \cap \textbf{PIE}$$

4. Assemble a policy.

$$pol = \textbf{if } d_1 \&\& \; d_2 \; \&\& \; .... \; d_n \textbf{ then } (explicit \sqcup implicit)$$

5. Update the dataflow graph.

$$\Gamma' = \Gamma[\, z \mapsto pol \,]$$

Implementation and Validation

We prototyped an implementation that analyses Java 1.4 programs

- Implementation (35,100 lines of Java code) extends the Polyglot compiler framework. [Nystrom, Clarkson, & Myers '03]
- Program analyses refine security inference:
  - Object-sensitive pointer analysis reasons precisely about non-aliased objects, preventing label creep.
  - Post dominance analysis determines identifies where *pc* levels may be safely lowered.
  - Precise type and exception analysis simplifies control flow graph by ruling out spurious exceptions (e.g. from casts that always succeed).

Case studies: successful inference of interesting information flows

Computer psychiatrist	Inputs sanitized before display.
Interactive password manager	Passwords only revealed via (simulated) cryptography.
Access-controlled key-value store	Information release governed by reference monitor.
Go Fish	The computer doesn't cheat.
Battleship game	The computer doesn't cheat. [Jif]
Mental poker	Private keys influence outputs appropriately [Askarov & Sabelfeld '05]

Sample policy: Battleship

```
        return lookup(key).getReader();
      }
      return null;
    }

    public String secretOf(String key) {
      if (lookup(key) != null) {
        return lookup(key).getData();
      }
      return null;
    }
}
class DbEntry{
    private String key;
    private Role reader;
    private String data;

    public DbEntry(String key, Role reader, String data){
      this.key = key;
      this.reader = reader;
      this.data = data;
    }

    public String getKey(){ return this.key; }
    public Role getReader(){ return this.reader; }
    public String getData(){ return this.data; }
}
class Login {
    private static class PwdTableEntry{
      String uname;
      String pwd;
      Role role;
      private PwdTableEntry(String uname, String pwd, Role role){
        this.uname=uname;
        this.pwd=pwd;
        this.role=role;
      }
    }
    private static SimpleArrayList pwds;

    public static void init() {
      pwds = new SimpleArrayList();
      pwds.add(new PwdTableEntry("Bond", "007", Role.SUPER_SPY));
      pwds.add(new PwdTableEntry("M", "0", Role.SPY));
      pwds.add(new PwdTableEntry("Q", "123", Role.TECH_GUY));
    }

    public static Role doLogin (String userName, String pwd){
      String id = "";
      for (int i=0; i< pwds.size(); i++){
        if (pwds.get(i) instanceof PwdTableEntry) {
          PwdTableEntry e = (PwdTableEntry) pwds.get(i);
          if( e.uname.equals(userName) && e.pwd.equals(pwd) ){
            return e.role;
          }
        }
      }
      return null;
    }
}
class Role {

    public static final Role SPY = new Role("spy");
    public static final Role MOLE = new Role("mole");
    public static final Role TECH_GUY = new Role("tech");

    private final String id;
    private Role(String id){
      this.id = id;
    }

    public static boolean lteq(Role r, Role s){
      if (r == null || s == null){
        return false;
      }
      if (r == SUPER_SPY) {
        return true;
      }
      if (r == MOLE) {
        return true;
      }
      return r.id.equals(s.id);
    }

}
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SimpleArrayList implements Iterable {

    List contents;

    SimpleArrayList() {this.contents = new Nil();}
    int size() { return contents.size(); }
    boolean isEmpty() {return contents.isEmpty(); }
    void clear() { this.contents = new Nil(); }

    private static abstract class List {
        private List() { }
        abstract int size();
        abstract boolean isEmpty();
        public abstract Object get(int i);
        public abstract List removePersistent(int i);
        public abstract void set(int i, Object o);
        public abstract Object head() throws NoSuchElementException;
        public abstract List tail() throws NoSuchElementException;
    }

    private static class Nil extends List {
        private Nil(){};
        public int size() { return 0; }
        public boolean isEmpty() { return true; }
        public Object get(int i) { throw new IllegalArgumentException(); }
        public List removePersistent(int i) {
            throw new IllegalArgumentException(); }

        public void set(int i, Object o) { throw new IllegalArgumentExcepti
        }
        public Object head() throws NoSuchElementException {
            throw new NoSuchElementException();
        }
        public Object head() throws NoSuchElementException {
            return lookup(key).getReader
        }
        return null;
        }

    public String secretOf(String key)
      if (lookup(key) != null) {
        return lookup(key).getData();
      }
      return null;
    }
}
class DbEntry{
    private String key;
    private Role reader;
    private String data;

    public DbEntry(String key, Role re
      this.key = key;
      this.reader = reader;
      this.data = data;
    }

    public String getKey(){ return this.
    public Role getReader(){ return th
    public String getData(){ return this
}
class Login {
    private static class PwdTableEntry
      String uname;
```

Sample policy: Battleship

Human player, p1

@input "p1 board"

Computer player, p2

Sample policy: Battleship

Human player, p1

Computer player, p2

@input "p1 board"

p2 query ↦ **if** (p2 query ok[0])
then Reveal(p2 query ok[1+], p2 hit p1?[0+], p1 hit p2?[0+])

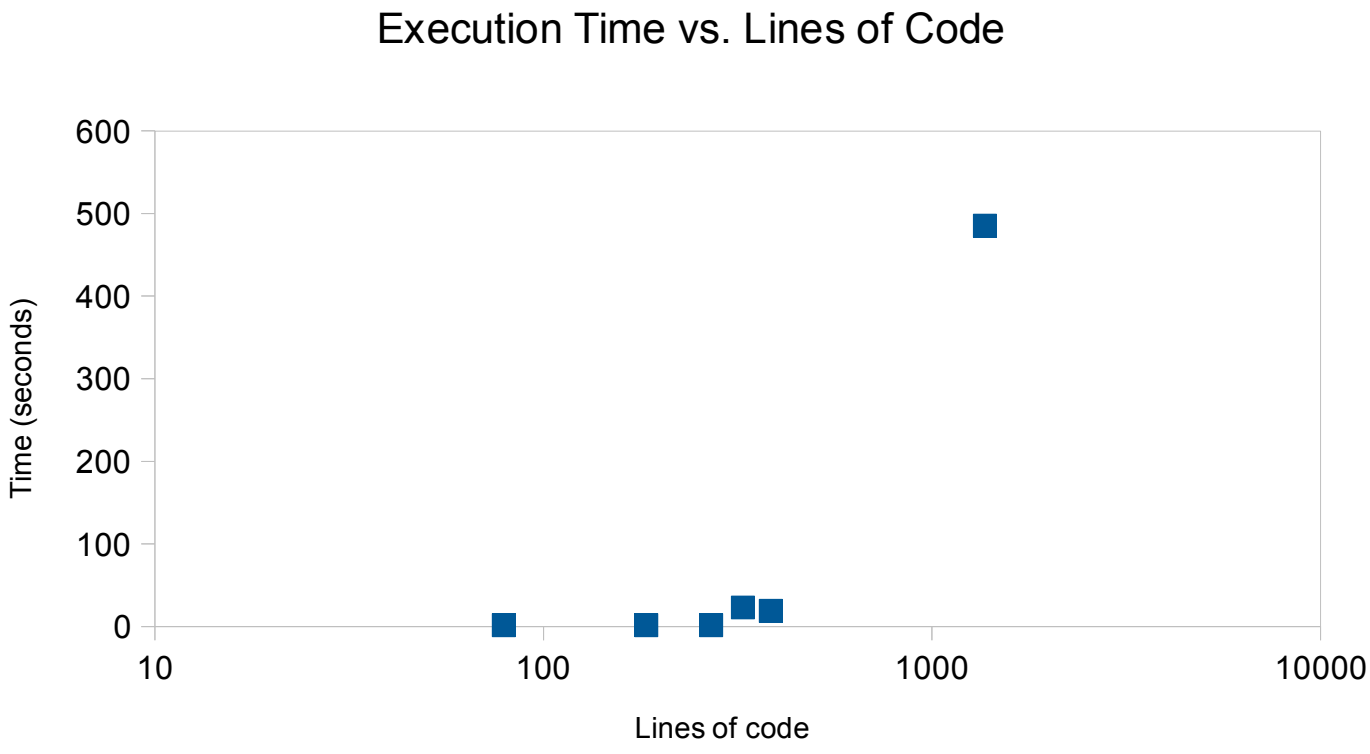
Computer (p2) player's queries only depend on public data.

Sample policy: Computer psychiatrist

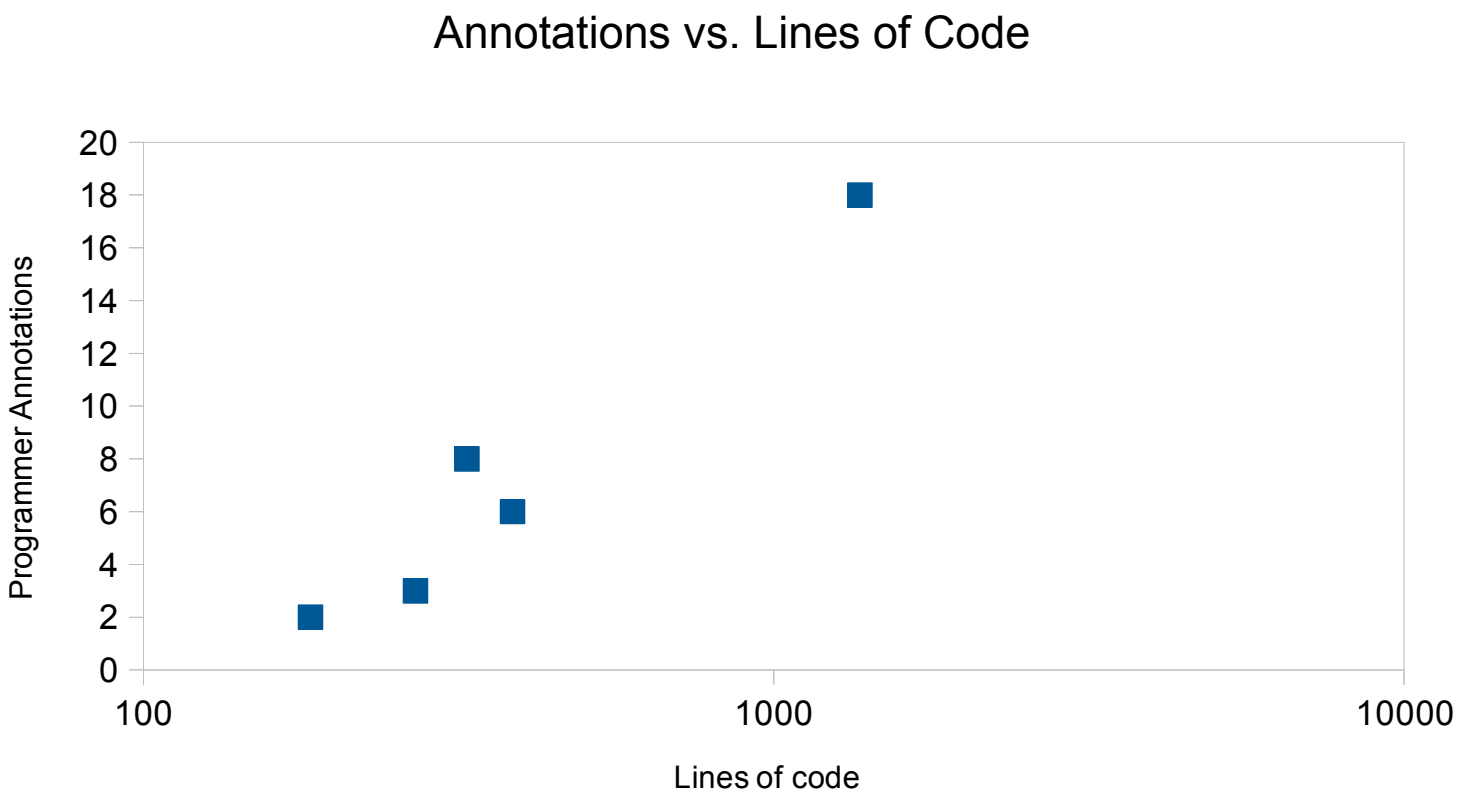
display $\mapsto$ **if-executed** (Liz.sanitize(java.lang.String))
then Reveal(**input**[0], **input**[1+])

Inputs are not displayed unless sanitize has been called.

The prototype scales to low 1000s loc, but is not performance tuned



We succeeded in requiring few programmer annotations



Conclusion

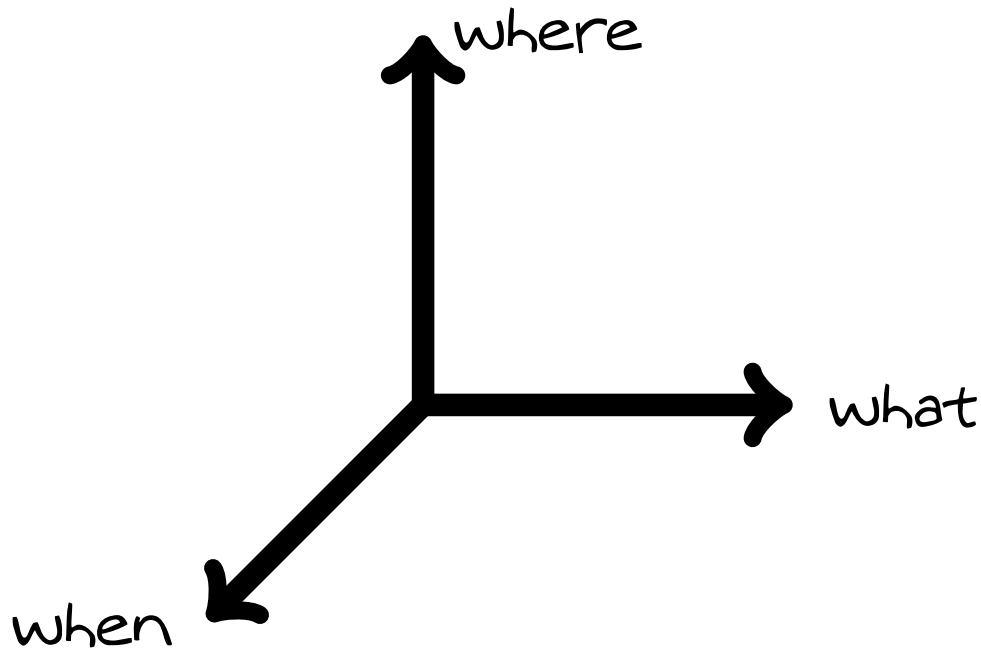
Policy inference via dataflow is a promising info. flow analysis

This project demonstrates:

- A novel workflow of iterative policy refinement via inference
- Expressive policies that explain information flows and reflect program structure
- A precise analysis suitable for (mostly) unmodified Java programs

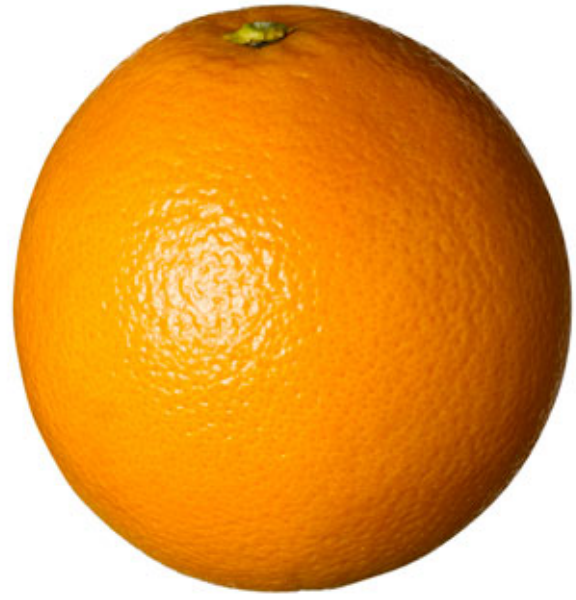
Bonus Slides

Policies express how information flows from inputs to outputs



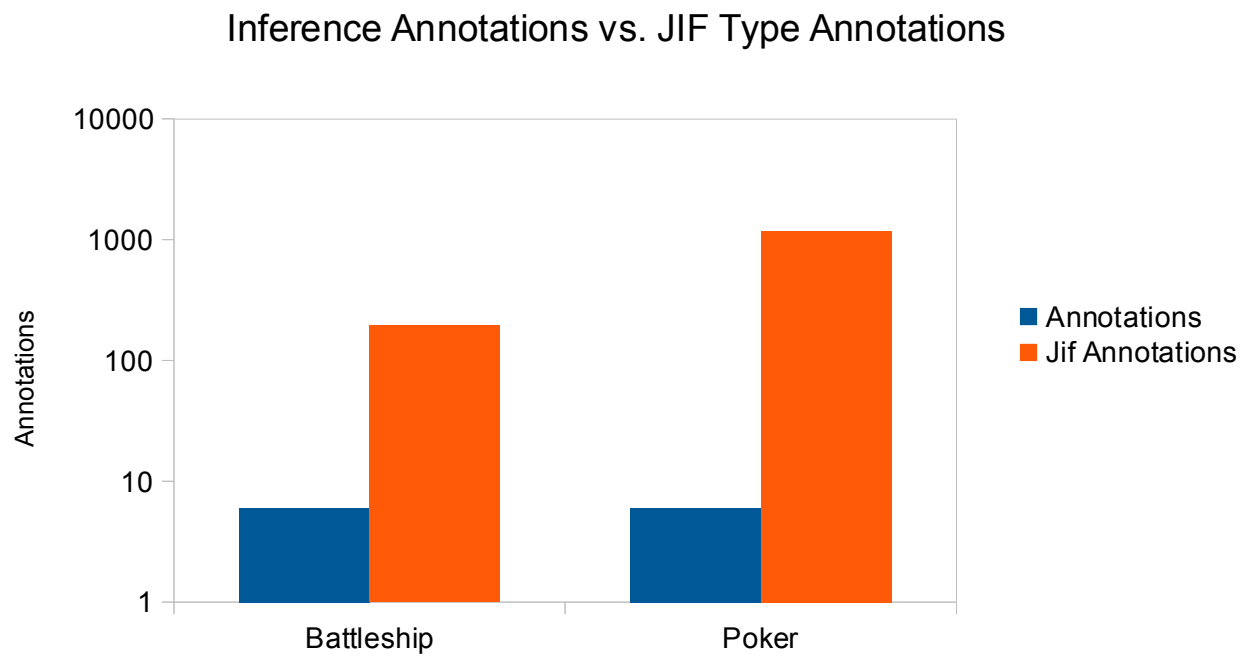
[Sabelfeld & Sands '05] NB. Our policies generalize delimited release [Askarov & Sabelfeld '07]

Annotation burden compares well with more explicit systems



Jif primary goals: modular typing, DLM policies, and not inference.
[Myers, Zheng, Zdancewic, Chong, Nystrom ’01–’09]

Annotation burden compares well with more explicit systems



Jif primary goals: modular typing, DLM policies, and not inference.
[Myers, Zheng, Zdancewic, Chong, Nystrom '01–'09]

Goal: Policies that explain and reflect information flows

Inferred policy

out $\mapsto$ **if** (authCheckOk[0])
then Reveal(secret[0+], authCheckOk[1+])

Policies...

- ...reflect structure of information flow within program
- ...describe flow in terms of meaningful *channel names*

Policy rewriting is needed for widening and improves readability

Example

$$\textbf{Reveal}(\mathsf{H}[j],\mathsf{H}[i+]) \;=\; \textbf{Reveal}[i+] \qquad (\text{provided } i \leq j)$$

$$\textbf{Reveal}(\mathsf{H}[j],\mathsf{H}[i+]) \;\sqsubseteq\; \textbf{Reveal}(\mathsf{H}[\min(i,j)+]) \;\sqsubseteq\; \textbf{Reveal}(\mathsf{H}[0+])$$

Widening $\nabla(p_1, p_2)$:

- Let $p := p_1$ **and** p_2
- If $|p| \leq \mathit{threshold}$ return p
- Rewrite $(=)$ until $|p| \leq \mathit{threshold}$ or no further rewrites possible
- Rewrite conservatively $(\sqsubseteq)$ until $|p| \leq \mathit{threshold}$
- Result $:= p$