

# Inference of Expressive Declassification Policies

Jeff Vaughan    Stephen Chong

UCLA



IEEE Security and Privacy

May 23, 2011



```

class Client {
    public static String takeReadingArgs(int i)
        throws java.util.NoSuchElementException {
        if(i<42) {
            return "book book";
        } else {
            throw new java.util.NoSuchElementException();
        }
    }

    public static void main(String args[]){
        String uname;
        String pwd;
        String key;
        try {
            uname = takeReadingArgs(0);
            pwd = takeReadingArgs(1);
            key = takeReadingArgs(2);
        } catch (Exception e){
            System.out.println(
                "Usage: java Client <uname> <pwd> <key>");
        }

        Login info();
        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.secretOfItemOut.println(
                    );

                *Access not authorizedOut.println(
                    );
            }

            private static boolean authorizedToRead(Role r, String key, Database db)
                (Role req)requires {
            }

            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;
                }

                private DbEntry(String key, Role reader,
                    String data;
                public DbEntry(String key, Role reader, String
                    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;
                    public List removePersistent(int i) {
                        throw new IllegalArgumentException();
                    }
                }

                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 (pwd.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 hasQ(Role r, Role s) {
                        if (r == null || s == null) {
                            return false;
                        }
                        if (s == SUPER_SPY {
                            return true;
                        }
                    }
                }
            }
        }
    }
}

```

# Inferred policy

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

```

        return true;
    }

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

    return next.removePersistent(i-1);
}

public void set(int i, Object o) {
    if(i < 0) {
        throw new IllegalArgumentException();
    }
}

import java.util.Iterator;

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 Cons(Object c, List l) {
        this.contents = c;
        this.next = l;
    }

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

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

        private class SimpleArrayListIterator implements Iterator {
            private List list;

            public SimpleArrayListIterator(SimpleArrayList simpleArrayList) {
                list = simpleArrayList.contents;
            }

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

            public Object next() {
                if(list instanceof Cons) {
                    Cons c = (Cons) list;
                    list = c.tail();
                    return c.head();
                }
                throw new NoSuchElementException();
            }

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

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

```

```

class Client {
    public static String takeReadingArg(int i)
        throws java.util.NoSuchElementException {
        if(i < 42) {
            return "book book";
        } else {
            throw new java.util.NoSuchElementException();
        }
    }

    public static void main(String args[]) {
        String username;
        String pwd;
        Stack key;

        System.out.println(
            "Usage: java Client <uname> <pwd> <days>");
        return;
    }

    @input "authCheckOk"
    @input "secret"
    @input "secret"
    @input "secret"
    @input "secret"

    private DbEntry login(String username, String pwd)
        throws java.util.NoSuchElementException {
        if(username == null || pwd == null)
            return null;

        Role role = new Role("spy");
        Role sPY = new Role("007");
        Role M = new Role("role");
        Role TECH_GUY = new Role("tech");

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

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

        public boolean isEmpty() {
            return true;
        }

        public Object get(int i) {
            if (i == 0)
                return this.contents;
            return next.removePersistent() - 1;
        }

        public void set(int i, Object o) {
            if (i < 0)
                throw new IllegalArgumentException();
        }

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

        private class SimpleArrayListIterator implements Iterator {
            private List list;

            public SimpleArrayListIterator(SimpleArrayList simpleArrayList) {
                list = simpleArrayList.contents;
            }

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

            public Object next() {
                if (list instanceof Cons) {
                    Cons c = (Cons) list;
                    list = c.tail();
                    return c.head();
                }
                throw new NoSuchElementException();
            }

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

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

        public boolean isEmpty() {
            return false;
        }

        public List removePersistent(int i) {
            if (i < 0)
                throw new IllegalArgumentException();
        }
    }
}

```

# Inferred policy

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

```

public boolean isEmpty() {
    return true;
}

public Object get(int i) {
    if (i == 0)
        return this.contents;
    return next.removePersistent() - 1;
}

public void set(int i, Object o) {
    if (i < 0)
        throw new IllegalArgumentException();
}

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

private class SimpleArrayListIterator implements Iterator {
    private List list;

    public SimpleArrayListIterator(SimpleArrayList simpleArrayList) {
        list = simpleArrayList.contents;
    }

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

    public Object next() {
        if (list instanceof Cons) {
            Cons c = (Cons) list;
            list = c.tail();
            return c.head();
        }
        throw new NoSuchElementException();
    }

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

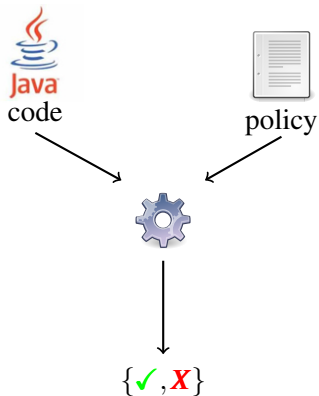
    public Iterator iterator() {
        return new SimpleArrayListIterator(this);
    }
}

public boolean isEmpty() {
    return false;
}

public List removePersistent(int i) {
    if (i < 0)
        throw new IllegalArgumentException();
}

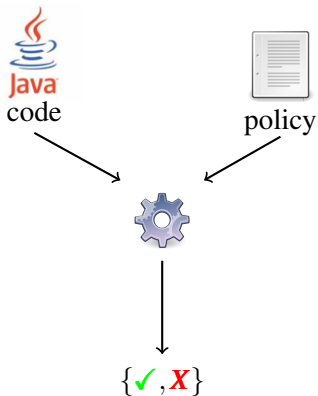
```

## Goal: Workflow of iterative policy refinement via inference

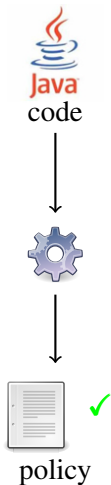


JIF [Myers et. al. '03],  
FlowCaml [Simonet '03], ...

# Goal: Workflow of iterative policy refinement via inference



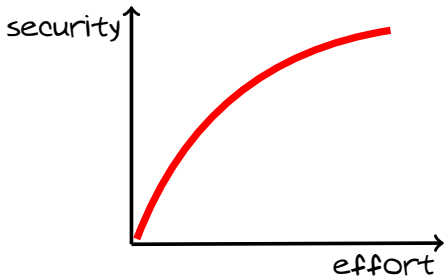
JIF [Myers et. al. '03],  
FlowCaml [Simonet '03], ...



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

**1** Policies

**2** Inference

**3** Implementation and Validation

## 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

⋮

## 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

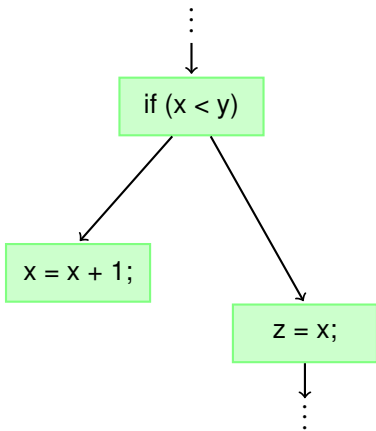
## Inference algorithm is based on dataflow analysis

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

## Inference algorithm is based on dataflow analysis

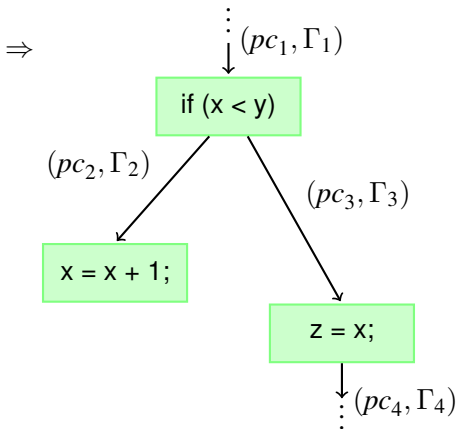
```
if (x < y){  
  x = x + 1;  
}  
z = x;
```

⇒



## Inference algorithm is based on dataflow analysis

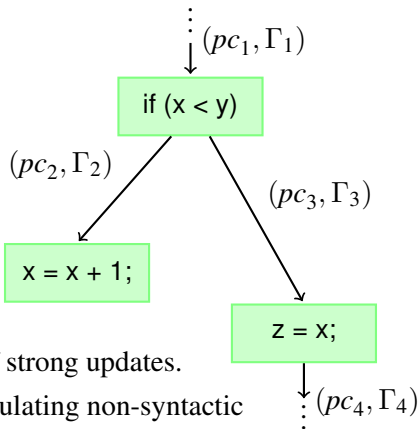
```
if (x < y){  
  x = x + 1;  
}  
z = x;
```



## Inference algorithm is based on dataflow analysis

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

$\Rightarrow$



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

## Contexts $(pc, \Gamma)$ approximates information flow

**Location context:** information content of fields or heap locations.

$$\Gamma : \mathbf{Location} \rightarrow \mathbf{Policy}$$

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

$$pc : \mathbf{BranchId} \rightarrow \mathbf{Policy} \cup \mathbf{PIE}$$

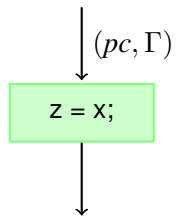
**Location** = abstract vars and refs

**PIE** = precise input exps.

**BranchId** = static branch ids

**Policy** = policies

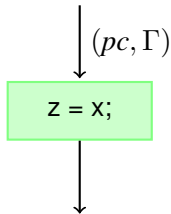
## Example: Updating a context for an assignment node



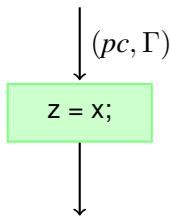
## Example: Updating a context for an assignment node

- 1 Calculate explicit flows.

$$\textit{explicit} = \Gamma(x)$$



## Example: Updating a context for an assignment node



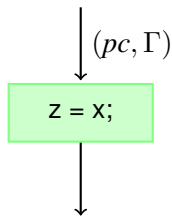
- 1 Calculate explicit flows.

$$explicit = \Gamma(x)$$

- 2 Calculate implicit, imprecise flows.

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

## Example: Updating a context for an assignment node



- 1 Calculate explicit flows.

$$explicit = \Gamma(x)$$

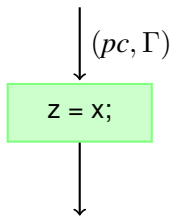
- 2 Calculate implicit, imprecise flows.

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

- 3 Calculate dominating precise input exps.

$$\{d_1, d_2, \dots, d_n\} = range(pc) \cap \mathbf{PIE}$$

## Example: Updating a context for an assignment node



- 1 Calculate explicit flows.

$$explicit = \Gamma(x)$$

- 2 Calculate implicit, imprecise flows.

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

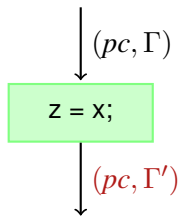
- 3 Calculate dominating precise input exps.

$$\{d_1, d_2, \dots, d_n\} = range(pc) \cap \mathbf{PIE}$$

- 4 Assemble a policy.

$$pol = \mathbf{if} \ d_1 \ \&\& \ d_2 \ \&\& \ \dots \ d_n \ \mathbf{then} \ (explicit \sqcup \ implicit)$$

## Example: Updating a context for an assignment node



- 1 Calculate explicit flows.

$$explicit = \Gamma(x)$$

- 2 Calculate implicit, imprecise flows.

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

- 3 Calculate dominating precise input exps.

$$\{d_1, d_2, \dots, d_n\} = range(pc) \cap \mathbf{PIE}$$

- 4 Assemble a policy.

$$pol = \mathbf{if} \ d_1 \ \&\& \ d_2 \ \&\& \ \dots \ d_n \ \mathbf{then} \ (explicit \sqcup \ implicit)$$

- 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;

        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", "007", Role.FEB_SQU));
        }

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

    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 == MOLE || s == MOLE) {
            return true;
        }
        if (r == TECH_GUY || s == TECH_GUY) {
            return true;
        }
        return false;
    }

    public static Role doLogin (String userName, String pwd) {
        if (r == MOLE) {
            return true;
        }
        if (r == TECH_GUY) {
            return true;
        }
        return false;
    }

    private static abstract class List {
        private List() {}
        abstract int size();
        abstract boolean isEmpty();
        abstract void clear() { this.contents = new Nil(); }

        private static abstract class List {
            public String secretOf(String key) {
                if (lookup(key) != null) {
                    return lookup(key).getData();
                }
                return null;
            }

            public abstract Object get(int i);
            public abstract void set(int i, Object o);
            public abstract Object head() throws NoSuchElementException;
            public abstract List tail() throws NoSuchElementException;
        }

        private String key;
        private Role reader;
        private String data;

        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 NoSuchElementException(); }
            public List removePersistent(int i) {
                throw new IllegalArgumentException();
            }
        }

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

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

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

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

    public class SimpleArrayList implements Iterable {
        List contents;

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

```

# Sample policy: Battleship

```

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

public String secretOf(String key) {
    if (lookup(key) != null) {
        return lookup(key).getData();
    }
    return null;
}

```



Human  
player, p1

```

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 Role getReader() { return this.reader; }
    public String getData() { return this.data; }
}

class Login {
    private static class PwdTableEntry{
        String username;

```

```

private PwdTableEntry(String username, String role) {
    this.username=username;
    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", "1", Role.FEB_SPY));
}

public static Role doLogin (§
    String username, String data) {
    if (pwds.size() > 0) {
        for (PwdTableEntry e = (PwdTableEntry) pwds.get(i) ; i < pwds.size(); i++) {
            if (e.username.equals(username) && e.pwd.equals(data)) {
                return e.role;
            }
        }
    }
    return null;
}

```



@input "p1 board"

```

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 == Role.SUPER_SPY || s == Role.SUPER_SPY) {
        return true;
    }
    if (r.pwd.equals(s.pwd)) {
        return true;
    }
    if (r.pwd.equals(s.pwd)) {
        return true;
    }
}

public class SimpleArrayList implements Iterable{
    List contents;

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

```



Computer  
player, p2

```

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

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

    public abstract Object get(int i);
    public abstract List removePersistent(int i);
    public abstract List set(int i, Object o);
    public abstract Object head() throws NoSuchElementException;
    public abstract List tail() throws NoSuchElementException;
}

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 Role getReader() { return this.reader; }
    public String getData() { return this.data; }
}

class Login {
    private static class PwdTableEntry{
        String username;

```

# 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 Role getReader() { return this.reader; }
    public String getData() { return this.data; }
}

class Login {
    private static class PwdTableEntry {
        String uname;
    }

    private static final Role SPY = new Role("spy");
}

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("C", "1234567890", Role.FBI_SPY));
}

public static Role doLogin(String uname, String pwd) {
    SimpleArrayList e = (SimpleArrayList) pwds;
    for (PwdTableEntry e : e) {
        if (e.uname.equals(uname) && e.pwd.equals(pwd)) {
            return e.role;
        }
    }
    return null;
}

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;
    }
    return r.getId().compareTo(s.getId()) <= 0;
}

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

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

    abstract Object get(int i);
    abstract List removePersistent(int i);
    abstract List set(int i, Object o);
    abstract Object head() throws NoSuchElementException;
    abstract List tail() throws NoSuchElementException;
}

class Nil extends List {
    private Role reader;
    private String key;
    private String data;

    public Nil() {
        this.reader = reader;
        this.key = key;
        this.data = data;
    }

    public boolean isEmpty() { return true; }
    public Object get(int i) { throw new NoSuchElementException(); }
    public List removePersistent(int i) {
        throw new IllegalArgumentException();
    }
    public String getKey() { return this.key; }
    public Role getReader() { return this.reader; }
    public String getData() { return this.data; }
}

class SimpleArrayList implements Iterable {
    List contents;
}

public class SimpleArrayList implements Iterable {
    List contents;
}

public static final Role SPY = new Role("spy");

```



Human player, p1



Computer player, p2

@input "p1 board"

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

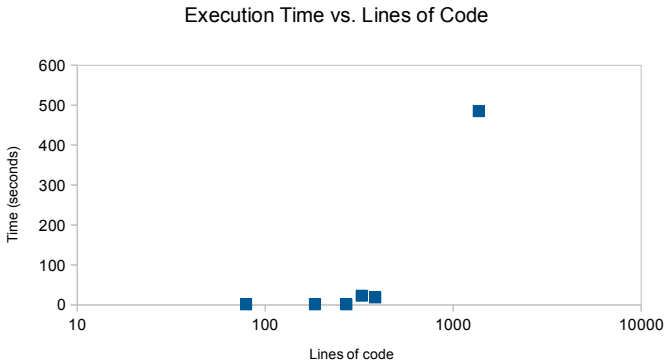
Computer (p2) player's queries only depend on public data. 19/24 UCLA

## Sample policy: Computer psychiatrist

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

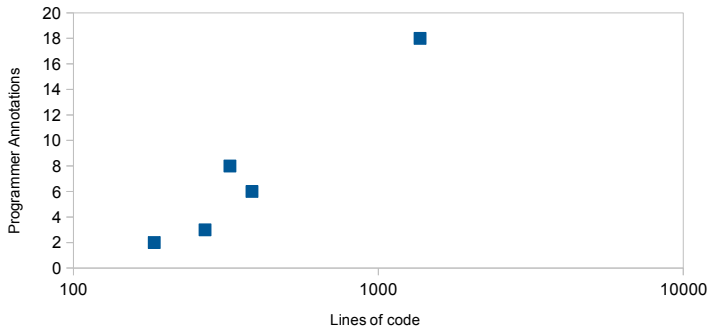
Inputs are not displayed unless **sanitize** has been called.

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



# We succeeded in requiring few programmer annotations

Annotations vs. Lines of Code



# Conclusion

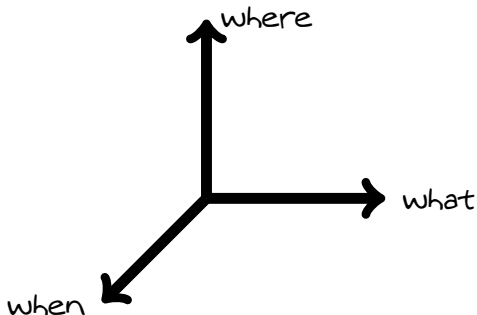
# Policy inference via dataflow is a promising info. flow analysis

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

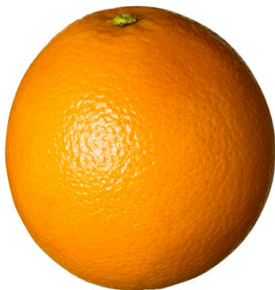
# Bonus Slides

## Policies express how information flows from inputs to outputs



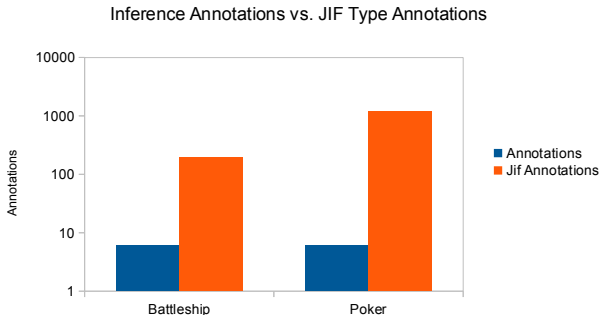
[Sabelfeld & Sands '05] NB. Our policies generalize **delimited release** [Askarov & Sabelfeld '07]

## 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

**Reveal**(H[j],H[i+]) = **Reveal**[i+] (provided  $i \leq j$ )

**Reveal**(H[j],H[i+])  $\sqsubseteq$  **Reveal**(H[min(i,j)+])  $\sqsubseteq$  **Reveal**(H[0+])

Widening  $\nabla(p_1, p_2)$ :

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