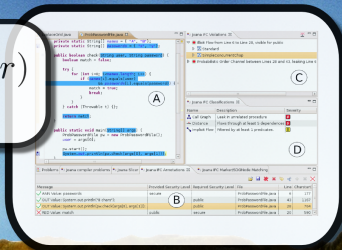


X-Rays, not Passport Checks – Information Flow Control Using JOANA

Gregor Snelting

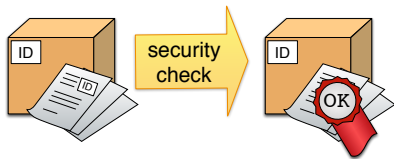
Presentation at SAP, 14.5.2014

$$\sum_{r \in \mathcal{I}} P_t(r) = \sum_{r \in \mathcal{U}} P_u(r)$$



Classical IT Security is not Enough!

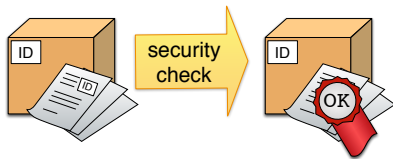
- classics: cryptography, certificates, intrusion detection, ... still necessary, but insufficient!
- classical approaches never analyse program code



- like passport checks – but passports can be faked

Classical IT Security is not Enough!

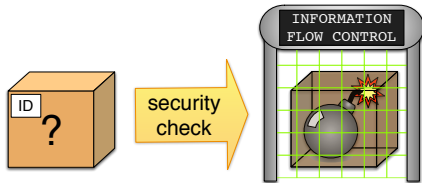
- classics: cryptography, certificates, intrusion detection, ... still necessary, but insufficient!
- classical approaches never analyse program code



- like passport checks – but passports can be faked
Example 1: Stuxnet used stolen certificates
Example 2: Heartbleed is based on an IFC problem

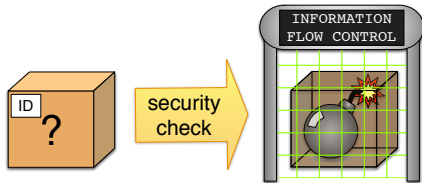
X-Rays, not Passport Checks!

- **Information Flow Control**: analyse source / machine code, uncovers leaks and illegal information flow



X-Rays, not Passport Checks!

- **Information Flow Control**: analyse source / machine code, uncovers leaks and illegal information flow

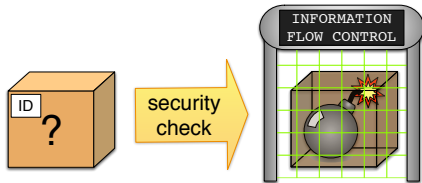



- advanced international research. Big projects: Mobius (EU), DFG SPP 1496 “Reliably Secure Software Systems”



X-Rays, not Passport Checks!

- **Information Flow Control**: analyse source / machine code, uncovers leaks and illegal information flow



- advanced international research. Big projects: Mobius (EU), DFG SPP 1496 “Reliably Secure Software Systems” 
- today: a few (!) useable tools

JOANA: Information Flow Control for Java
Download: joana.ipd.kit.edu



Information Flow Control (IFC)

IFC analyses source/byte code, guarantees:

confidentiality: secret (“high”) values do not flow to public (“low”) ports

integrity: critical (“high”) computations not manipulated from outside (“low”)

Information Flow Control (IFC)

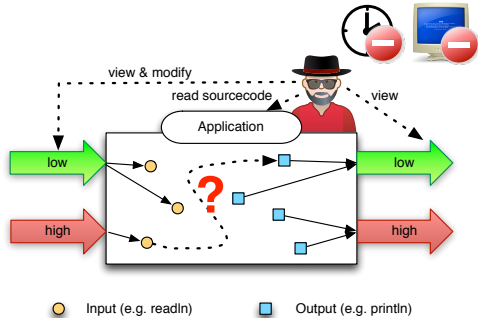
IFC analyses source/byte code, guarantees:

confidentiality: secret (“high”) values do not flow to public (“low”) ports

integrity: critical (“high”) computations not manipulated from outside (“low”)

Assumptions:

- compiler, OS, hardware, ... are secure. IFC checks only application code!
- attacker knows code, can observe public output
- no physical side channels!



Confidentiality Leaks

attacker gathers information about secret PIN:

```
void main():  
    // inputPIN is high  
    // print is low  
    x = inputPIN();  
    if (x < 1234)  
        print(0);  
    y = x;  
    print(y);
```

explicit/implicit leaks

data or control flow depend
on PIN

Confidentiality Leaks

attacker gathers information about secret PIN:

```
void main():  
    // inputPIN is high  
    // print is low  
    x = inputPIN();  
    if (x < 1234)  
        print(0);  
    y = x;  
    print(y);
```

explicit/implicit leaks

data or control flow depend
on PIN

```
void thread_1():  
    // input is low  
    x = input();  
    print(x);  
  
void thread_2():  
    y = inputPIN();  
    x = y;
```

possibilistic leak

some interleavings leak PIN

Confidentiality Leaks

attacker gathers information about secret PIN:

```
void main():  
    // inputPIN is high  
    // print is low  
    x = inputPIN();  
    if (x < 1234)  
        print(0);  
    y = x;  
    print(y);
```

explicit/implicit leaks

data or control flow depend
on PIN

```
void thread_1():  
    // input is low  
    x = input();  
    print(x);  
  
void thread_2():  
    y = inputPIN();  
    x = y;
```

possibilistic leak

some interleavings leak PIN

```
void thread_1():  
    print("SA");  
  
void thread_2():  
    y = inputPIN();  
    while (y != 0)  
        y--;  
    print("P");
```

probabilistic leak

Confidentiality Leaks

attacker gathers information about secret PIN:

```
void main():  
    // inputPIN is high  
    // print is low  
    x = inputPIN();  
    if (x < 1234)  
        print(0);  
    y = x;  
    print(y);
```

explicit/implicit leaks

data or control flow depend on PIN

```
void thread_1():  
    // input is low  
    x = input();  
    print(x);  
  
void thread_2():  
    y = inputPIN();  
    x = y;
```

possibilistic leak

some interleavings leak PIN

```
void thread_1():  
    print("SA");  
  
void thread_2():  
    y = inputPIN();  
    while (y != 0)  
        y--;  
    print("P");
```

probabilistic leak

$P(\text{"SAP"})$ depends on PIN

IFC Technology

- theoretical security notion: (probabilistic) noninterference
- analysis methods: type systems, model checking, PDGs, ...

- theoretical security notion: (probabilistic) noninterference
- analysis methods: type systems, model checking, PDGs, ...

Quality criteria:

- **sound** IFC guarantees to find **all** leaks!
soundness proof [machine checked] required
- **precise** IFC generates few false alarms!
sophisticated analysis algorithms required

- theoretical security notion: (probabilistic) noninterference
- analysis methods: type systems, model checking, PDGs, ...

Quality criteria:

- **sound** IFC guarantees to find **all** leaks!
soundness proof [machine checked] required
- **precise** IFC generates few false alarms!
sophisticated analysis algorithms required
Remember **Rice's Theorem**: 100% sound **and** precise program analysis is **undecidable**

- theoretical security notion: (probabilistic) noninterference
- analysis methods: type systems, model checking, PDGs, ...

Quality criteria:

- **sound** IFC guarantees to find **all** leaks!
soundness proof [machine checked] required
- **precise** IFC generates few false alarms!
sophisticated analysis algorithms required
Remember **Rice's Theorem**: 100% sound **and** precise program analysis is **undecidable**
- **scaleable** IFC analyses big programs!
algorithm engineering required
- **full-range** IFC analyses full Java / C# / C++ !
pointer analysis infrastructure required
- **useable** IFC needs little preprocessing!
few annotations & nice GUI required



IFC Tools

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise 📉
- TAJ / Andromeda [Pistoia et al. 2009]: static analysis (part of IBM Security AppScan); full Java, high scalability, BUT moderately precise

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise 📉
- TAJ / Andromeda [Pistoia et al. 2009]: static analysis (part of IBM Security AppScan); full Java, high scalability, BUT moderately precise 👍
- TaintDroid [Enck et al. 2010]: dynamic analysis; full Java, Android, application studies, BUT unsound, explicit flows (“taint”) only

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise 📉
- TAJ / Andromeda [Pistoia et al. 2009]: static analysis (part of IBM Security AppScan); full Java, high scalability, BUT moderately precise 👍
- TaintDroid [Enck et al. 2010]: dynamic analysis; full Java, Android, application studies, BUT unsound, explicit flows (“taint”) only 👍
- FlowDroid [Bodden 2013]: static analysis; no implicit flows, no probabilistic leaks, unsound, BUT Android apps & lifecycle

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise 📉
- TAJ / Andromeda [Pistoia et al. 2009]: static analysis (part of IBM Security AppScan); full Java, high scalability, BUT moderately precise 👍
- TaintDroid [Enck et al. 2010]: dynamic analysis; full Java, Android, application studies, BUT unsound, explicit flows (“taint”) only 👍
- FlowDroid [Bodden 2013]: static analysis; no implicit flows, no probabilistic leaks, unsound, BUT Android apps & lifecycle 👍

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise 📉
- TAJ / Andromeda [Pistoia et al. 2009]: static analysis (part of IBM Security AppScan); full Java, high scalability, BUT moderately precise 👍
- TaintDroid [Enck et al. 2010]: dynamic analysis; full Java, Android, application studies, BUT unsound, explicit flows (“taint”) only 👍
- FlowDroid [Bodden 2013]: static analysis; no implicit flows, no probabilistic leaks, unsound, BUT Android apps & lifecycle 👍
- JOANA: static analysis; see below 👍

- JIF [Myers et al 99]: static analysis; special language, many annotations, unprecise 📉
- TAJ / Andromeda [Pistoia et al. 2009]: static analysis (part of IBM Security AppScan); full Java, high scalability, BUT moderately precise 👍
- TaintDroid [Enck et al. 2010]: dynamic analysis; full Java, Android, application studies, BUT unsound, explicit flows (“taint”) only 👍
- FlowDroid [Bodden 2013]: static analysis; no implicit flows, no probabilistic leaks, unsound, BUT Android apps & lifecycle 👍
- JOANA: static analysis; see below 👍

Do not confuse IFC tools with bug-finding tools (ESC/Java, Clousot, ...) !

- IFC tools find leaks, bug finders find null pointers, missing locks, ... many bug finders are scaleable (MLoc), but very unsound!

Noninterference

- basic idea: public output is not influenced by secret data!
- **sequential noninterference**: for program Q , for all initial states s, s'

$$s \sim_{low} s' \implies \llbracket Q \rrbracket s \sim_{low} \llbracket Q \rrbracket s'$$

- basic idea: public output is not influenced by secret data!
- **sequential noninterference**: for program Q , for all initial states s, s'

$$s \sim_{low} s' \implies \llbracket Q \rrbracket s \sim_{low} \llbracket Q \rrbracket s'$$

- for concurrent programs: treatment of nondeterminism?!
idea: *probability* of public outputs is not influenced by secret data

- basic idea: public output is not influenced by secret data!
- **sequential noninterference**: for program Q , for all initial states s, s'

$$s \sim_{low} s' \implies \llbracket Q \rrbracket s \sim_{low} \llbracket Q \rrbracket s'$$

- for concurrent programs: treatment of nondeterminism?!
idea: *probability* of public outputs is not influenced by secret data
- Q is **probabilistic noninterferent** if

$$\sum_{t \in \mathfrak{I}} P_i(t) = \sum_{t \in \mathfrak{I}} P_{i'}(t)$$

where $P_i(t)$ is the probability of trace t under input i , \mathfrak{I} are the low-equivalent traces caused by i

JOANA in a Nutshell

```

01 void main() {
02   int h = input();
03   int l = encode(h);
04   output(l);
05 }
06
07 int encode(int x) {
08   if (x > 42)
09     return 1;
10   else
11     return 0;
12 }
    
```



+

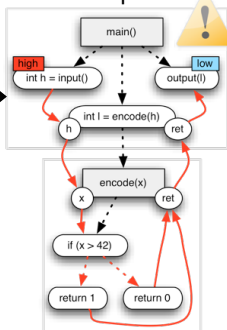
security
lattice



annotations



System Dependence Graph



- full Java (up to 100kLOC)
- static whole program analysis
- applies program slicing
- applies points-to analysis
- flow-, context-, object-sensitive
- threads: probabilistic & possibilistic

Analysis Result

non-interference
guarantee
or
possible leaks



Machine-checked proofs

- Classical non-interference with slicing
- Slicing theorem
 - \exists path $a \rightarrow b$
 \Rightarrow definitely no information flow
 - \exists path $a \rightarrow b$
 \Rightarrow information flow possible



JOANA Features

- sound
- full Java bytecode
- unlimited threads
- few false alarms
- few annotations
- declassifications
- Android Apps
- Eclipse plugin, webstart GUI
- open source



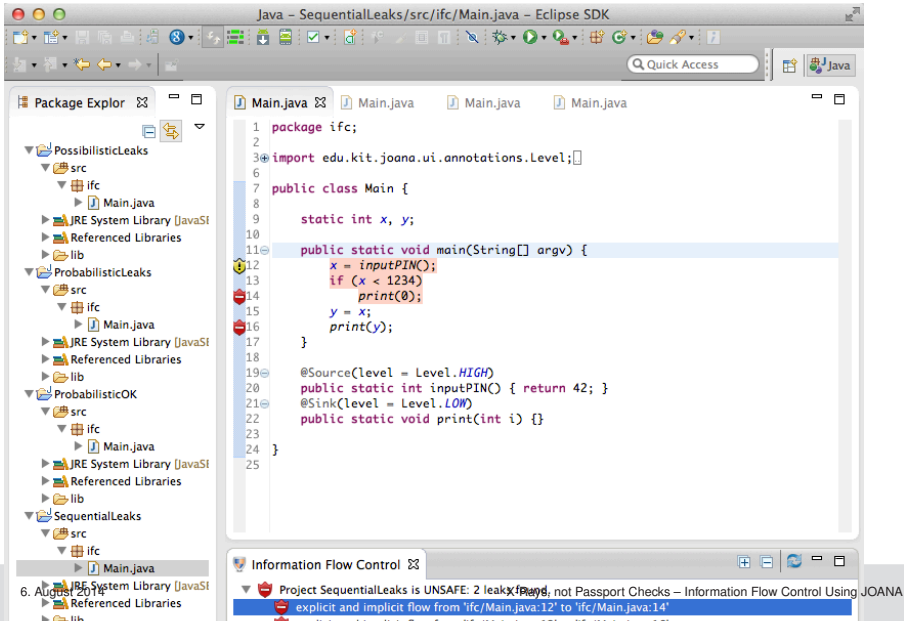
JOANA Features

- sound
- full Java bytecode
- unlimited threads
- few false alarms
- few annotations
- declassifications
- Android Apps
- Eclipse plugin, webstart GUI
- open source
- max 100kLoc
- case studies
 - e.g. HSQLDB (50kLOC Java): analysis time \approx 1 day on PC
- **scenario**: analyse security kernels / critical components, not full OS!



Jürgen Graf: Analysis of sequential & probabilistic leaks

Implicit Leak

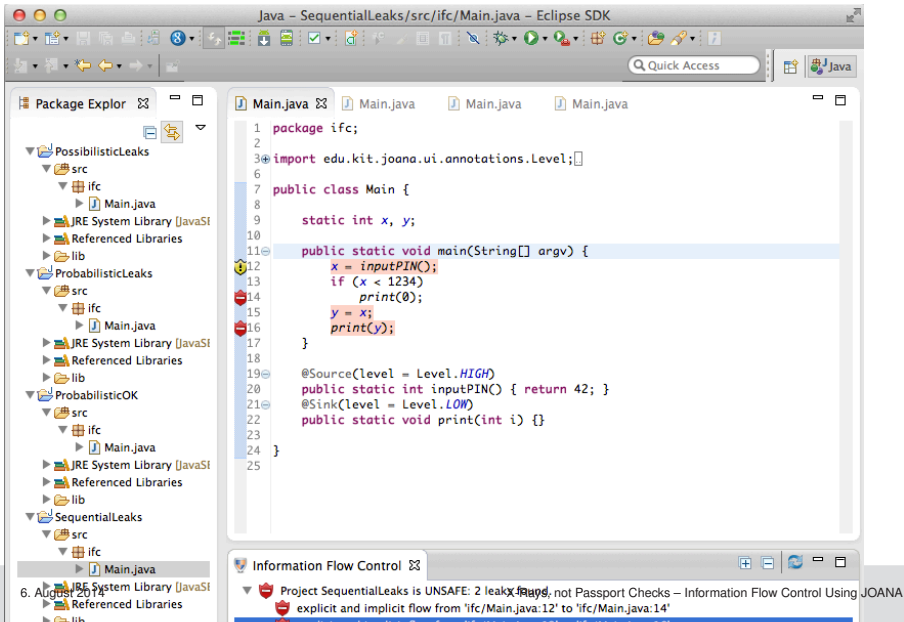


The screenshot shows the Eclipse IDE interface. The Package Explorer on the left shows a project structure with several packages: PossibilisticLeaks, ProbabilisticLeaks, ProbabilisticOK, and SequentialLeaks. The Main.java file in the SequentialLeaks package is open in the editor. The code is as follows:

```
1 package ifc;
2
3 import edu.kit.joana.ui.annotations.Level;
4
5
6
7 public class Main {
8
9     static int x, y;
10
11     public static void main(String[] argv) {
12         x = inputPIN();
13         if (x < 1234)
14             print(0);
15         y = x;
16         print(y);
17     }
18
19     @Source(level = Level.HIGH)
20     public static int inputPIN() { return 42; }
21     @Sink(level = Level.LOW)
22     public static void print(int i) {}
23
24 }
25
```

The Information Flow Control (IFC) view at the bottom shows a warning: "Project SequentialLeaks is UNSAFE: 2 leaks found, not Passport Checks – Information Flow Control Using JOANA". A red arrow points to a specific leak: "explicit and implicit flow from 'ifc/Main.java:12' to 'ifc/Main.java:14'".

Explicit Leak

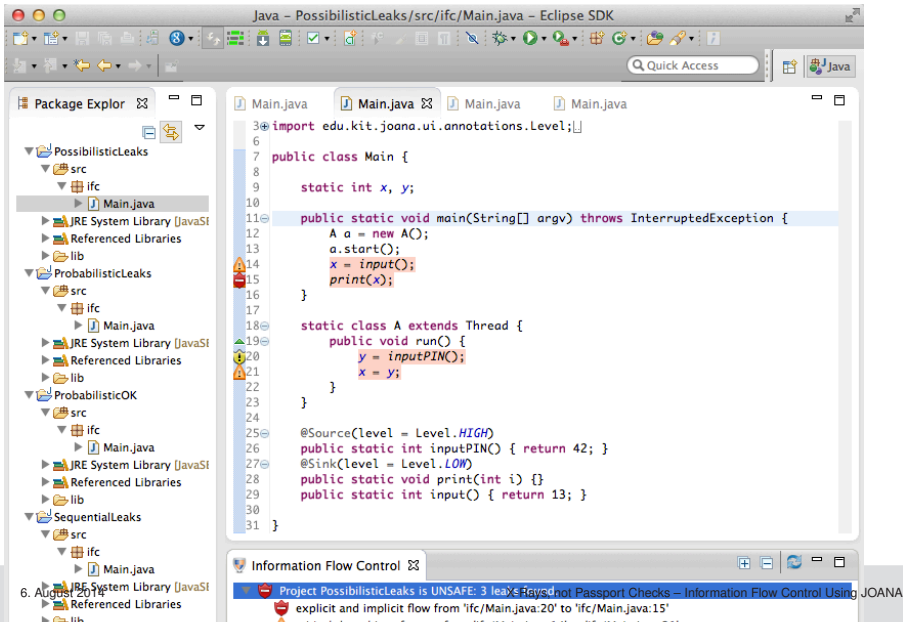


The screenshot shows the Eclipse IDE interface. The Package Explorer on the left shows a project structure with folders for ProbabilisticLeaks, ProbabilisticOK, and SequentialLeaks, each containing a src folder with an ifc folder and a Main.java file. The Main.java file is open in the editor, showing the following code:

```
1 package ifc;
2
3 import edu.kit.joana.ui.annotations.Level;
4
5
6
7 public class Main {
8
9     static int x, y;
10
11     public static void main(String[] argv) {
12         x = inputPIN();
13         if (x < 1234)
14             print(0);
15         y = x;
16         print(y);
17     }
18
19     @Source(level = Level.HIGH)
20     public static int inputPIN() { return 42; }
21     @Sink(level = Level.LOW)
22     public static void print(int i) {}
23
24 }
25
```

The Information Flow Control (IFC) warning at the bottom indicates a security issue: "Project SequentialLeaks is UNSAFE: 2 leaks found, not Passport Checks – Information Flow Control Using JOANA explicit and implicit flow from 'ifc/Main.java:12' to 'ifc/Main.java:14'".

Possibilistic Leak



Java – PossibilisticLeaks/src/ifc/Main.java – Eclipse SDK

Package Explorer

- PossibilisticLeaks
 - src
 - ifc
 - Main.java
- ProbabilisticLeaks
 - src
 - ifc
 - Main.java
- ProbabilisticOK
 - src
 - ifc
 - Main.java
- SequentialLeaks
 - src
 - ifc
 - Main.java

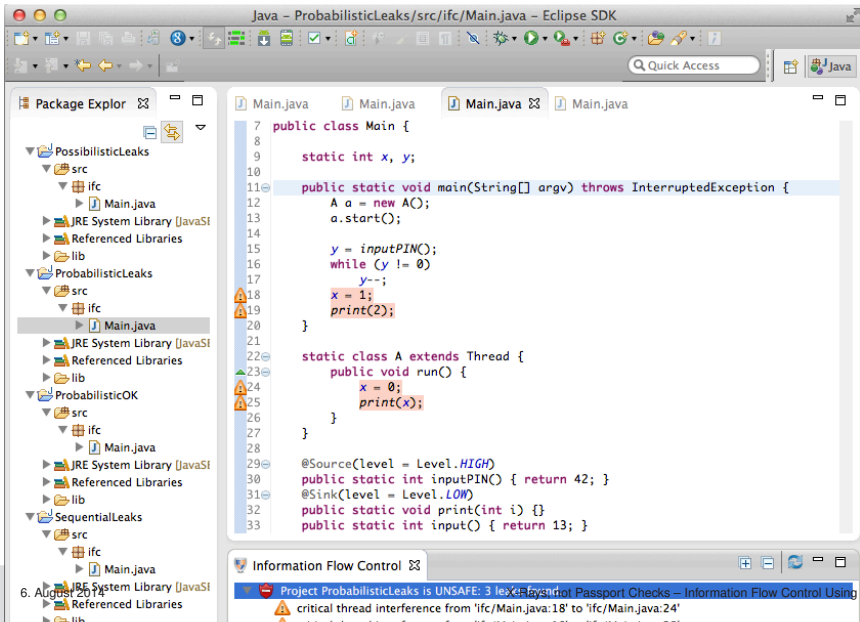
```
3 import edu.kit.joana.ui.annotations.Level;
6
7 public class Main {
8
9     static int x, y;
10
11     public static void main(String[] argv) throws InterruptedException {
12         A a = new A();
13         a.start();
14         x = input();
15         print(x);
16     }
17
18     static class A extends Thread {
19         public void run() {
20             y = inputPIN();
21             x = y;
22         }
23     }
24
25     @Source(level = Level.HIGH)
26     public static int inputPIN() { return 42; }
27     @Sink(level = Level.LOW)
28     public static void print(int i) {}
29     public static int input() { return 13; }
30
31 }
```

Information Flow Control

Project PossibilisticLeaks is UNSAFE: 3 leaks. Reason: not Passport Checks – Information Flow Control Using JOANA

explicit and implicit flow from 'ifc/Main.java:20' to 'ifc/Main.java:15'

Probabilistic Leak



Java – ProbabilisticLeaks/src/ifc/Main.java – Eclipse SDK

Package Explorer

- ProbabilisticLeaks
 - src
 - ifc
 - Main.java

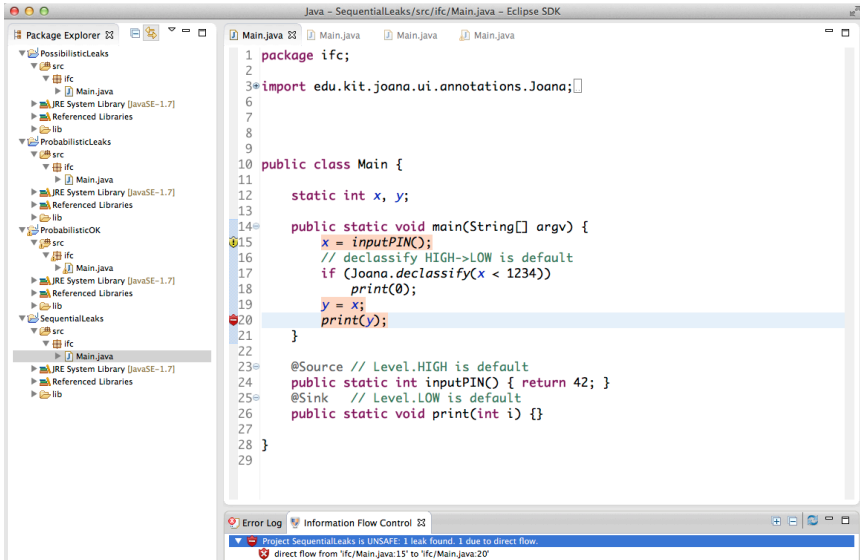
```
7 public class Main {
8
9     static int x, y;
10
11     public static void main(String[] argv) throws InterruptedException {
12         A a = new A();
13         a.start();
14
15         y = inputPIN();
16         while (y != 0)
17             y--;
18         x = 1;
19         print(2);
20     }
21
22     static class A extends Thread {
23         public void run() {
24             x = 0;
25             print(x);
26         }
27     }
28
29     @Source(level = Level.HIGH)
30     public static int inputPIN() { return 42; }
31     @Sink(level = Level.LOW)
32     public static void print(int i) {}
33     public static int input() { return 13; }
```

Information Flow Control

Project ProbabilisticLeaks is UNSAFE: 3 leaks found

critical thread interference from 'ifc/Main.java:18' to 'ifc/Main.java:24'

Declassification



The screenshot shows the Eclipse IDE interface. The Package Explorer on the left shows a project structure with folders for 'PossibilisticLeaks', 'ProbabilisticLeaks', and 'SequentialLeaks'. The 'SequentialLeaks' folder is expanded to show 'src' and 'ifc' subfolders, with 'Main.java' selected. The main editor window displays the code for 'Main.java' in the 'ifc' package. The code includes a package declaration, an import for 'edu.kit.joana.ui.annotations.Joana', and a 'Main' class with a 'main' method. The 'main' method contains logic for declassification based on a PIN value. Line 20, 'print(y);', is highlighted in blue. The bottom status bar shows an error log with the message: 'Project SequentialLeaks is UNSAFE: 1 leak found. 1 due to direct flow. direct flow from 'ifc/Main.java:15' to 'ifc/Main.java:20''.

```
1 package ifc;
2
3 import edu.kit.joana.ui.annotations.Joana;
4
5
6
7
8
9
10 public class Main {
11
12     static int x, y;
13
14     public static void main(String[] argv) {
15         x = inputPIN();
16         // declassify HIGH->LOW is default
17         if (Joana.declassify(x < 1234))
18             print(0);
19         y = x;
20         print(y);
21     }
22
23     @Source // Level.HIGH is default
24     public static int inputPIN() { return 42; }
25     @Sink // Level.LOW is default
26     public static void print(int i) {}
27
28 }
29
```

Error Log: Information Flow Control

- Project SequentialLeaks is UNSAFE: 1 leak found. 1 due to direct flow.
- direct flow from 'ifc/Main.java:15' to 'ifc/Main.java:20'

- based on sophisticated program analysis:
program dependence graphs (PDGs); exception-, pointer-, ... -analysis
- flow-, context-, object-, field-sensitive; optionally time-, lock-sensitive
⇒ high precision, few false alarms

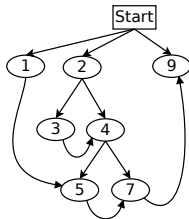
- based on sophisticated program analysis:
program dependence graphs (PDGs); exception-, pointer-, ... -analysis
- flow-, context-, object-, field-sensitive; optionally time-, lock-sensitive
⇒ high precision, few false alarms
- (sequential) declassification in case noninterference is too strict
- machine-checked soundness proofs for sequential IFC

- based on sophisticated program analysis:
program dependence graphs (PDGs); exception-, pointer-, ... -analysis
- flow-, context-, object-, field-sensitive; optionally time-, lock-sensitive
⇒ high precision, few false alarms
- (sequential) declassification in case noninterference is too strict
- machine-checked soundness proofs for sequential IFC
- for concurrent programs: new **RLSOD** algorithm
[Relaxed Low-Security Observable Determinism]
⇒ probabilistic noninterference without previous restrictions

A small PDG

```

1 a = u();
2 while (f()) {
3   x = v();
4   if (x>0)
5     b = a;
6   else
7     c = b;
8 }
9 z = c;
```



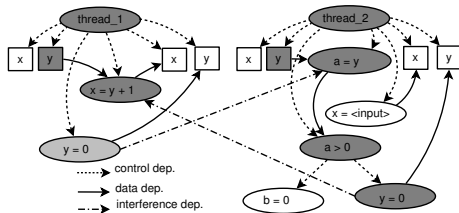
- $x \rightarrow y$: x controls execution of y ; $x \rightsquigarrow y$: assigned var in x is used in y
- **backward slice** $BS(x) = \{y \mid y \rightarrow^* x\}$
- **Slicing Theorem.** [Reps et al 1988]
Only statements/ expressions $y \in BS(x)$ can influence behaviour at x
- $u()$ can influence z , a cannot influence $x > 0$
- PDGs for full Java are **nontrivial**
25 years of international research!

A multi-threaded PDG

```
int x, y;

void thread_1():
  x = y + 1;
  y = 0;

void thread_2():
  a = y;
  x = <input>;
  if a > 0
    b = 0;
  else
    y = 0;
```



- $BS(x) = \{y \mid y \xrightarrow{*}_{realizable} x\}$
“realizable”: context- time- object-sensitive
black: $BS("x = y + 1;")$; grey: time insensitive
- **Theorem.**[Snelting et al 2006] A program is (sequentially) noninterferent, if no high source is in backward slice of a low sink
machine-checked proof: [Wasserrab 2009]

Conclusion

- IFC today is practical: [X-rays](#), not passport checks
- JOANA offers precise IFC for realistic Java programs
- JOANA contains groundbreaking algorithms + validation + proofs
- JOANA is open source
- JOANA was used in realistic case studies

Conclusion

- IFC today is practical: *X-rays, not passport checks*
- JOANA offers precise IFC for realistic Java programs
- JOANA contains groundbreaking algorithms + validation + proofs
- JOANA is open source
- JOANA was used in realistic case studies
- **new**: JOANA handles pluggable (Android) components
- **new**: JOANA handles message encryption without declassification

Conclusion

- IFC today is practical: X-rays, not passport checks
- JOANA offers precise IFC for realistic Java programs
- JOANA contains groundbreaking algorithms + validation + proofs
- JOANA is open source
- JOANA was used in realistic case studies
- **new**: JOANA handles pluggable (Android) components
- **new**: JOANA handles message encryption without declassification

JOANA is an achievement in IT security

JOANA main contributors:

G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, J. Krinke, M. Mohr, D. Wasserrab

JOANA sponsors: DFG Sn11/5-1/2, DFG Sn11/9-1/2, DFG Sn11/11-1/2, DFG Sn11/12-1/2 [SPP 1496 “Reliably Secure Software Systems”], BMBF Center for Cyber Security KASTEL

JOANA papers: TOSEM 2006, IJIS 2009, PLAS 2009, CSF 2012, IT 2014, IJIS 2014, ...

Low-Security Observational Determinism [Roscoe] [Zdancewicz]:
low-equivalent inputs must generate low-equivalent traces

- $i \sim_{low} i'$, \mathcal{T} possible traces for i , \mathcal{U} possible traces for i'
 $\implies \forall T, U \in \mathcal{T} \cup \mathcal{U} : T \sim_{low} U$

“the order of low events is not influenced by high events”

⇒ LSOD is scheduler independent

Theorem. [Zdancewic 2003]

LSOD guarantees probabilistic noninterference

Low-Security Observational Determinism [Roscoe] [Zdancewicz]:
low-equivalent inputs must generate low-equivalent traces

- $i \sim_{low} i'$, \mathcal{T} possible traces for i , \mathcal{U} possible traces for i'
 $\implies \forall T, U \in \mathcal{T} \cup \mathcal{U} : T \sim_{low} U$

“the order of low events is not influenced by high events”

\Rightarrow LSOD is scheduler independent

Theorem. [Zdancewic 2003]

LSOD guarantees probabilistic noninterference

- **BUT** soundness problems / severe restrictions in early LSOD definitions
 \Rightarrow so far, other approaches more popular: Weak probabilistic noninterference [Volpano&Smith], Strong security [Sabelfeld&Sands], ...

NEW: RLSOD

Relaxed LSOD [Giffhorn 2012PhD, Giffhorn & Snelting 2013]:

- guarantees probabilistic noninterference
- avoids prohibition of secure low-nondeterminism
- precise: flow- context- object- field- time-sensitive
- soundness proof
- full Java, arbitrary threads (no reflection)
- scales up to 100kLOC
- succesful case studies [Küsters & Graf 2012, ...]

NEW: RLSOD

Relaxed LSOD [Giffhorn 2012PhD, Giffhorn & Snelting 2013]:

- guarantees probabilistic noninterference
- avoids prohibition of secure low-nondeterminism
- precise: flow- context- object- field- time-sensitive
- soundness proof
- full Java, arbitrary threads (no reflection)
- scales up to 100kLOC
- succesful case studies [Küsters & Graf 2012, ...]

Flow-sensitivity is the key! other ingredients:

- new definition for $T \sim_{low} U$ in case of nontermination
⇒ no soundness leaks for infinite traces
cave: RLSOD is termination-insensitive
- uses [program dependence graphs](#) (PDGs)
⇒ sound & precise static approximation of RLSOD criterion

- so far, IFC cannot handle crypto (e.g. encrypted message passing)
IFC needs declassification for crypto channels !?

⇒ Küster's idea [CSF 2012]:

1. replace crypto code by stub which generates random numbers: $P \rightsquigarrow P'$
2. use JOANA to prove that P' is secure
3. Theorem: if P' secure, and P uses “perfect” crypto, then P secure
 (“noninterference guarantees computational indistinguishability w.r.t. unbounded adversaries”)

⇒ allows to apply JOANA to distributed systems, where components communicate via encrypted messages: e-voting, cloud storage

- recent work: Integration with KeY, extend for digital signatures and symmetric crypto (“CVJ” Projekt)