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

Gregor Snelting

Presentation at SAP, 14.5.2014
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 a buffer over-read problem
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.

  ![Diagram](image.png)

  - ID
  - Security check
  - Information Flow Control

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!

[Diagram showing input and output channels with confidentiality and integrity aspects]
Confidentiality Leaks

attacker gathers information about secret PIN:

```c
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
```

```c
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 ("SAP"") depends on PIN
```
Confidentiality Leaks

attacker gathers information about secret PIN:

```c
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

```c
void thread_1():
    // input is low
    x = input();
    print(x);
```

```c
void thread_2():
    y = inputPIN();
    x = y;
```

**possibilistic leak**
some interleavings leak PIN

```c
void thread_1():
    print("SA");
```

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

Confidentiality Leaks

attacker gathers information about secret PIN:

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

void thread_1():
    // input is low
    x = input();
    print(x);

void thread_2():
    y = inputPIN();
    x = y;
```

explicit/implicit leaks
data or control flow depend on PIN

possibilistic leak
some interleavings leak PIN

probabilistic leak

P (SAP) depends on PIN
Confidentiality Leaks

attacker gathers information about secret PIN:

```c
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

```c
void thread_1():
   // input is low
   x = input();
   print(x);
```

possibilistic leak
some interleavings leak PIN

```c
void thread_2():
   y = inputPIN();
   x = y;
```

probabilistic leak
\( P(“SAP“) \) depends on PIN

```c
void thread_1():
   print("SA");
```

```c
void thread_2():
   y = inputPIN();
   while (y != 0)
      y--;
   print("P");
```
IFC Technology

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

Remember Rice's Theorem: 100% sound and precise program analysis is undecidable.
IFC Technology

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

Quality criteria:

- sound IFC guarantees to find \textbf{all} leaks!
  soundness proof [machine checked] required
- precise IFC generates few false alarms!
  sophisticated analysis algorithms required
ICF Technology

- 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 undecideable
ICF Technology

- 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 undecideable
- 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
IFC Tools

- 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

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 scalable (MLoc), but very unsound!
IFC Tools

- 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

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!
IFC Tools

- 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

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!
IFC Tools

- 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 ✅

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!
IFC Tools

- 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!
IFC Tools

- 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 $s, s'$

$$s \sim_{\text{low}} s' \implies [Q]s \sim_{\text{low}} [Q]s'$$
Noninterference

- basic idea: public output is not influenced by secret data!
- sequential noninterference: for program $Q$, for all initial states $s, s'$
  \[ s \sim_{low} s' \implies [Q]s \sim_{low} [Q]s' \]
- for concurrent programs: treatment of nondeterminism?!
  idea: probability of public outputs is not influenced by secret data
Noninterference

- basic idea: public output is not influenced by secret data!
- sequential noninterference: for program $Q$, for all initial states $s, s'$
  \[
  s \sim_{low} s' \implies \llbracket Q \rrbracket s \sim_{low} \llbracket Q \rrbracket s'
  \]

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

- $Q$ is probabilistic noninterferent if
  \[
  \sum_{t \in \mathcal{T}} P_i(t) = \sum_{t \in \mathcal{U}} P_{i'}(t)
  \]

  where $P_i(t)$ is the probability of trace $t$ under input $i$, $\mathcal{T}$ are the low-equivalent traces caused by $i$
JOANA in a Nutshell

• Classical non-interference with slicing
• Slicing theorem

```
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

```
main()

input() -> high

encode(x) -> low

if (x > 42) -> return 1

return 0

return 1

return 0

output() -> low
```

Analysis Result

non-interference guarantee
or
possible leaks

Machine-checked proofs

• Classical non-interference with slicing
• Slicing theorem
  • exists path a → b
  • implies definitely no information flow
  • exists path a → b
  • implies 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 ≈ 1 day on PC

scenario: analyse security kernels / critical components, not full OS!
Jürgen Graf: Analysis of sequential & probabilistic leaks
Implicit Leak

```java
package ifc;
import edu.kit.joana.ui.annotations.Level;

public class Main {
    static int x, y;

    public static void main(String[] argv) {
        x = inputPIN();
        if (x < 1234)
            print(0);
        y = x;
        print(y);
    }

    @Source(level = Level.HIGH)
    public static int inputPIN() { return 42; }
    @Sink(level = Level.LOW)
    public static void print(int i) {}
}
```
Explicit Leak

```java
package ifc;
import edu.kit.joana.ui.annotations.Level;

public class Main {
    static int x, y;

    public static void main(String[] argv) {
        x = inputPIN();
        if (x < 1234)
            print(0);
        y = x;
        print(y);
    }

    @Source(level = Level.HIGH)
    public static int inputPIN() { return 42; }
    @Sink(level = Level.LOW)
    public static void print(int i) {}
}
```
Possibilistic Leak

---

```java
import edu.kit.ioa.ia.annotations.Level;

public class Main {
    static int x, y;

    public static void main(String[] argv) throws InterruptedException {
        A a = new A();
        a.start();
        x = input();
        print(x);
    }

    static class A extends Thread {
        public void run() {
            y = inputPIN();
            x = y;
        }
    }

    @Source(level = Level.HIGH)
    public static int inputPIN() { return 42; }

    @Sink(level = Level.LOW)
    public static void print(int i) {}

    public static int input() { return 13; }
}
```

---

19. Mai 2014

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

```java
public class Main {
    static int x, y;
    
    public static void main(String[] argv) throws InterruptedException {
        A a = new A();
        a.start();
        
        y = inputPIN();
        while (y != 0)
            y--;
        x = 1;
        print(2);
    
    static class A extends Thread {
        public void run() {
            x = 0;
            print(x);
        }
    
    @Source(level = Level.HIGH)
    public static int inputPIN() { return 42; }
    @Sink(level = Level.LOW)
    public static void print(int i) {};
    public static int input() { return 13; }
}
```
Declassification

package ifc;

import edu.kit.joana.ui.annotations.Joana;

public class Main {
    static int x, y;

    public static void main(String[] argv) {
        x = inputPIN();
        // declassify HIGH->LOW is default
        if (Joana.declassify(x < 1234))
            print(0);
        y = x;
        print(y);

        @Source // Level.HIGH is default
        public static int inputPIN() { return 42; }

        @Sink // Level.LOW is default
        public static void print(int i) {}
JOANA Technology

- 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

⇒ high precision, few false alarms

(machine-checked soundness proofs for sequential IFC)

⇒ probabilistic noninterference without previous restrictions
JOANA Technology

- 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
JOANA Technology

- 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. \( \text{while } (f()) \) {
3. \( x = v(); \)
4. \( \text{if } (x>0) \)
5. \( b = a; \)
6. \( \text{else} \)
7. \( c = b; \)
8. \}
9. \( z = c; \)

- \( x \rightarrow y \): \( x \) controls execution of \( y \); \( x \leadsto y \): assigned var in \( x \) is used in \( y \)
- **backward slice** \( BS(x) = \{ y \mid y \leadsto^* x \} \)
- **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

```c
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 | y \rightarrow^{\text{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

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

\[ i \sim_{\text{low}} i', \exists \text{ possible traces for } i, \exists \text{ possible traces for } i' \]
\[ \implies \forall T, U \in \mathcal{T} \cup \mathcal{U} : T \sim_{\text{low}} U \]

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

\[ \implies \text{LSOD is scheduler independent} \]

Theorem. [Zdancewic 2003]
LSOD guarantees probabilistic noninterference
Low-Security Observational Determinism [Roscoe] [Zdancewicz]:
low-equivalent inputs must generate low-equivalent traces

\[ i \sim_{\text{low}} i', \exists \text{ possible traces for } i, \exists \text{ possible traces for } i' \]
\[ \implies \forall T, U \in \mathcal{I} \cup \mathcal{U} : T \sim_{\text{low}} U \]

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

\[ \implies \text{LSOD is scheduler independent} \]

**Theorem.** [Zdancewic 2003]
LSOD guarantees probabilistic noninterference

**BUT** soundness problems / severe restrictions in early LSOD definitions
\[ \implies \text{so far, other approaches more popular: Weak probabilistic noninterference} \]
\[ [\text{Volpano\&Smith}], \text{Strong security} [\text{Sabelfeld\&Sands}], \ldots \]
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
- successful case studies [Küsters & Graf 2012, ...]

Flow-sensitivity is the key! other ingredients:
- new definition for $T \sim_{\text{low}} U$ 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
NEW: IFC and Crypto

- 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: $P \sim P'$
  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)