# Tool Demonstration: JOANA

Jürgen Graf(✉), Martin Hecker, Martin Mohr, and Gregor Snelting

Karlsruhe Institute of Technology, Karlsruhe, Germany
{graf,martin.hecker,martin.mohr,gregor.snelting}@kit.edu

**Abstract.** JOANA is a tool for information flow control, which can handle full Java with unlimited threads and scales to ca. 100 kLOC. JOANA uses a new algorithm for checking probabilistic noninterference, named RLSOD. JOANA uses a stack of sophisticated program analysis techniques which minimise false alarms. JOANA is open source (joana.ipd.kit.edu) and offers an Eclipse GUI as well as an API.

The current tool demonstration paper concentrates on JOANA's precision. Effects of flow-sensitivity, context-sensitivity, and object-sensitivity are explained, as well as precision gains from the new RLSOD criterion.

**Keywords:** Information flow control · Probabilistic noninterference · Program analysis

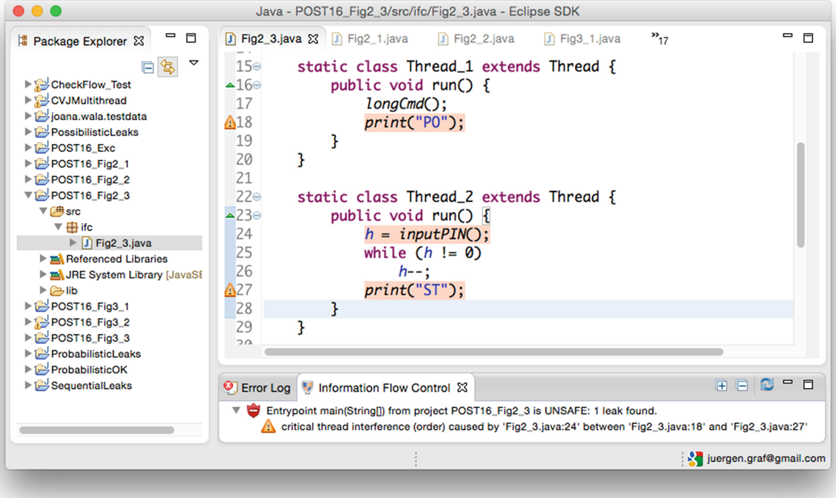
## 1 Introduction

JOANA is a tool for information flow control (IFC), which discovers all confidentiality and integrity leaks in Java programs. JOANA is open source (joana.ipd.kit.edu). In this tool demonstration paper, we concentrate on the precision of JOANA. JOANA is based on sophisticated program analysis (points-to analysis, exception analysis, program dependence graphs), can handle full Java with unlimited threads, and scales to ca. 100 kLOC. JOANA minimizes false alarms through flow- context- field- object- time- and lock-sensitive analysis algorithms. JOANA guarantees to find all explicit, implicit, possibilistic, and probabilistic leaks. The theoretical foundations have been described in [2,5,10]. The GUI and specific usage aspects have been described in [3,4,6–9].

Figure 1 shows the JOANA Eclipse plugin. In the source code, input and output are annotated with security levels ("High" i.e. secret, or "Low" i.e. public; only input and output need annotations). The program contains a probabilistic leak, because the running time of the loop depends on secret input, and thus the probability that "POST" is printed instead of "STPO" due to interleaving depends on secret input. The leak is highlighted in the source code (full details on a leak are available on demand). JOANA allows arbitrary security lattices (not just "High" and "Low"). It can check confidentiality as well as integrity; in this paper we concentrate on confidentiality.

JOANA analyses Java bytecode and uses IBM's WALA analysis frontend; recently, a frontend for Android bytecode was added. JOANA offers various

F. Piessens and L. Viganò (Eds.): POST 2016, LNCS 9635, pp. 89–93, 2016.
DOI: 10.1007/978-3-662-49635-0_5



```
15    static class Thread_1 extends Thread {
16        public void run() {
17            longCmd();
18            print("PO");
19        }
20    }
21
22    static class Thread_2 extends Thread {
23        public void run() {
24            h = inputPIN();
25            while (h != 0)
26                h--;
27            print("ST");
28        }
29    }
```

```
Entrypoint main(String[]) from project POST16_Fig2_3 is UNSAFE: 1 leak found.
    critical thread interference (order) caused by 'Fig2_3.java:24' between 'Fig2_3.java:18' and 'Fig2_3.java:27'
```

**Fig. 1.** JOANA GUI, discovering a probabilistic leak

options for analysis precision (e.g. object-sensitive points-to analysis, time-sensitive backward slicing). It was thus able to provide security guarantees for several examples from the literature which are considered difficult. More interesting is perhaps the successful analysis of an experimental e-voting system developed by Küsters et al. [7]. In a scalability study, the full source code of the HSQLDB database was analysed; analysis needed one day on a standard PC.

```
1  void main():
2    read(H);
3    if (H < 1234)
4      print(0);
5    L = H;
6    print(L);
```

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    read(L);
6    print(L);
7  void thread_2():
8    read(H);
9    L = H;
```

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    longCmd;
6    print("PO");
7  void thread_2():
8    read(H);
9    while (H != 0)
10     H--;
11   print("ST");
```

**Fig. 2.** Some leaks. Left: explicit and implicit, middle: possibilistic, right: probabilistic.

Figure 2 presents small but typical confidentiality leaks (as usual, H is “High”, L is “Low”). Explicit leaks arise if (parts of) secret values are copied (indirectly) to public output. Implicit leaks arise if a secret value can change control flow (which can change public behaviour). Possibilistic leaks in concurrent programs arise if a certain interleaving produces an explicit or implicit leak; in Fig. 2 middle, interleaving order 5,8,9,6 causes an explicit leak. Probabilistic leaks arise if the probability of public output is influenced be secret values; in Fig. 2 right, H is never copied to L, but if the value of H is large, probability is higher that “POST” is printed instead of “STPO”.

## 2 Sequential Precision

JOANA was the first IFC tool which used program dependence graphs (PDGs). PDGs are naturally flow- and context-sensitive. Today PDGs for (multi-threaded) full Java are highly precise and scale up to 1 MLOC. As a prerequisite for PDGs, a precise points-to analysis and exception analysis is necessary.

```
1 void main():
2   read(H);
3   L = 2;
4   H1 = f(H);
5   L1 = f(L);
6   print(L1);
7
8 int f(int x)
9   {return x+42;}
```

```
1   o1 = new O(); //O1
2   o2 = new O(); //O2
3   o1.c = H;
4   o2.c = L;
5   o3 = o2;
6   print(o3.c);
7   o4 = o1;
8   o4 = o2;
```

```
1 void main():
2   fork thread_1();
3   fork thread_2();
4 void thread_1():
5   L = 42;
6   read(H);
7 void thread_2():
8   print(L);
9   L = H;
```

**Fig. 3.** Unprecise analysis causes false alarms.

A flow-insensitive analysis will ignore statement order. In Fig. 3 right, a false alarm results if order of statements 8/9 is ignored. Flow-insensitive analysis also ignores killing definitions as in L=H; L=42; (where the second statement can be far from the first). A context-insensitive analysis will merge different calls to the same function. In Fig. 3 left, a context-insensitive analysis will merge the two calls to f and cause a false alarm. In practice, context-sensitivity is even more important for precision than flow-sensitivity. Object-sensitivity means that fields in different objects of the same class are distinguished. In Fig. 3 middle, an object-insensitive analysis will merge o1 and o2 and cause a false alarm. Object-sensitivity is difficult in case of nested or recursive object definitions, because it interferes with points-to analysis and requires additional field-sensitivity. For analysing object-oriented programs, the combination of context- and object-sensitivity is essential, as in o1.c=o1.f(H); o2.c=o1.f(L);.

Points-to analysis determines for every pointer a set of objects it may point to. In Fig. 3 middle, $pt(o1) = \{O1\}$, $pt(o3) = pt(o2) = \{O2\}, pt(o4) = \{O1, O2\}$. For precision, points-to sets should be small, but of course maintain soundness. Today, sophisticated points-to algorithms for full Java are known, however many common approaches are not efficient for PDG-based IFC. Hence, we are exploring *sweet-spots* of points-to analyses with a better precision-cost-ratio, which employ precision only where needed. The key is to taylor the points-to analysis to the concrete IFC query: First we build a PDG with an imprecise but cheap points-to analysis and compute a forward slice [5] of the high statements. From this slice, we extract the critical instances that may contain high information. In a second pass, we apply an automatically taylored points-to analysis which specifically distinguishes the critical instances.

Another issue is exception analysis, as exceptions can generate much additional control flow and spoil precision. We implemented a null pointer analysis which detects field accesses that never dereference null. We are currently implementing an analogous checker for array accesses.

## 3 Probabilistic Precision

Different criteria and algorithms for probabilistic noninterference (PN) have been proposed. The vast majority is not flow-sensitive (let alone context- or object-sensitive), or puts unacceptable restrictions on programs ("no low statements after high guards"); some turned out to be unsound. The simplest criterion, LSOD (low-security observational determinism) is scheduler independent and easy to implement as a program analysis. However LSOD strictly prohibits any, even secure, low-nondeterminism. For example Fig. 3 right is PN according to the original definition, but interleaving can cause low nondeterminism (in fact a race) for statements 5 and 8, hence LSOD causes a false alarm.

We found that flow-sensitivity is essential for a precise LSOD, and that LSOD can naturally be checked using PDGs for multi-threaded programs [2]. We then devised an improvement, RLSOD (relaxed LSOD), which allows low-nondeterminism if it cannot be reached from high events. The latter can be checked easily in the CFG. For example, Fig. 3 right is not LSOD but RLSOD, because the 5/8 race cannot be reached from high statements. Recently, we discovered that for low-nondeterministic statements $s_1, s_2$, it is enough to check high events in the control flow regions between the immediate dominator $idom(s_1, s_2)$, and $s1$ (resp. $s2$). For example assume that in `main` the initial statement `read(H)` is added. Then the 5/8 race *can* be reached from a high event, so "simple" RLSOD causes a false alarm. But the "dominator" improvement, plus flow-sensitivity (which respects the 8/9 order) will classify the example as secure – which it is according to PN.

Another issue is precision of the may-happen-in-parallel analysis, which is part of the RLSOD algorithm [2]. The current MHP analysis is context-sensitive but ignores explicit locks; recently, experiments with a lock-sensitive MHP based on pushdown networks [1] have begun.

## 4 Fine Tuning

JOANA supports a wide range of configuration options that allows experts to fine-tune the analysis. JOANA also comes with an automated approach to infer a reasonable configuration and additionally categorize detected leaks. Given a program and security annotations, JOANA starts with a fast but imprecise points-to analysis and subsequently applies more precise algorithms until either the maximal precision option has been reached or no leaks are detected.

JOANA helps to categorize the severity of detected information leaks as follows: (1) Flow through exceptions is ignored. Disappearing leaks are categorized as *caused by exceptions*. (2) Implicit flow through control dependencies is ignored. Leaks still detected are categorized as *explicit*, disappeared leaks as *implicit*. The result is a noninterference guarantee or a list of information leaks with source code location and categorization.

**Acknowledgements.** JOANA was partially supported by Deutsche Forschungsgemeinschaft in the scope of SPP "Reliably Secure Software Systems".

## References

1. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 199–213. Springer, Heidelberg (2011)
2. Giffhorn, D., Snelting, G.: A new algorithm for low-deterministic security. Int. J. Inf. Secur. **14**(3), 263–287 (2015)
3. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in Java programs - a practical guide. In: Proceedings of 6th Working Conference on Programming Languages (ATPS 2013). Lecture Notes in Informatics (LNI), vol. 215, pp. 123–138. Springer, Heidelberg (2013)
4. Graf, J., Hecker, M., Mohr, M., Snelting, G.: Checking applications using security APIs with JOANA. In: 8th International Workshop on Analysis of Security APIs, July 2015
5. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. **8**(6), 399–422 (2009)
6. Küsters, R., Scapin, E., Truderung, T., Graf, J.: Extending and applying a framework for the cryptographic verification of java programs. In: Abadi, M., Kremer, S. (eds.) POST 2014 (ETAPS 2014). LNCS, vol. 8414, pp. 220–239. Springer, Heidelberg (2014)
7. Küsters, R., Truderung, T., Graf, J.: A framework for the cryptographic verification of Java-like programs. In: 2012 IEEE 25th Computer Security Foundations Symposium (CSF). IEEE Computer Society, June 2012
8. Mohr, M., Graf, J., Hecker, M.: JoDroid: adding android support to a static information flow control tool. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering , Dresden, Germany, 17-18 Mäarz 2015, vol. 1337 of CEUR Workshop Proceedings, pp. 140–145. CEUR-WS.org (2015)
9. Snelting, G., Giffhorn, D., Graf, J., Hammer, C., Hecker, M., Wasserrab, D.: Checking probabilistic noninterference using JOANA. IT - Inf. Technol. **56**, 280–287 (2014)
10. Wasserrab, D., Lohner, D., Snelting, G.: On PDG-based noninterference and its modular proof. In: Proceedings of PLAS 2009. ACM, June 2009