Checking Probabilistic Noninterference Using JOANA

Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, Daniel Wasserrab

Abstract: JOANA is a tool for software security analysis, checking up to 100kLOC of full multi-threaded Java. JOANA is based on sophisticated program analysis techniques and thus very precise. It includes a new algorithm guaranteeing probabilistic noninterference, named RLSOD. JOANA needs few annotations and has a nice GUI. The tool is open source and was applied in several case studies. The article presents an overview of JOANA and its underlying technology.

ACM CCS: Security and privacy → Information flow control; Theory of computation → Program analysis; General and reference → Verification; Software and its engineering → Software verification and validation

Keywords: software security, program analysis, noninterference

1 Overview

Classical software security techniques, such as certificates, do not analyse the actual behaviour of programs and thus cannot provide guarantees about integrity and confidentiality of software. Information flow control (IFC) is an additional fine-grained analysis of software source or machine code, which uncovers all security leaks, or provides a true guarantee about integrity resp. confidentiality. IFC is typically based on some notion of noninterference, which demands that public behaviour is not influenced by secret data and thus guarantees confidentiality.[1] Many noninterference criteria have been proposed, and many IFC analysis algorithms constructed; these vary widely in features such as soundness ("all leaks guaranteed to be found"), precision ("no false alarms"), scalability ("big programs"), language ("full Java"), usability ("few annotations"), compositionality ("modular analysis"), and others.

JOANA is an IFC tool developed at KIT. JOANA is available for public download, or can be used by everybody through a Java webstart GUI.[2] The engineer must provide Java sources to be analysed, where all input and output statements are annotated "high" (secret) or "low" (public)[3] – other statements do not need annotations. JOANA can handle full Java bytecode with arbitrary threads, scales to ca. 100kLOC, and empirically demonstrated high precision [12, 11, 10]. JOANA is based on a stack of sophisticated program analysis algorithms (pointer analysis, exception analysis, program dependence graphs). JOANA minimizes false alarms through flow-, context-, object-, field-, time-, and lock-sensitive analysis techniques. JOANA allows declassification along sequential information flows. In concurrent programs, all possibilistic and probabilistic leaks are discovered. JOANA comes with a soundness proof; the soundness proof for the sequential part was machine-checked in Isabelle [31, 30]. JOANA was used in realistic case studies such as [18, 17]. Practical application is described in detail in [7].

In the following, we will summarize experiences with JOANA, and sketch the underlying technology. In-depth descriptions of the latter can be found in [12, 6].

2 Application of JOANA

Figure 1 shows the JOANA plugin for Eclipse. In the source code window, the full source for example (1) from Figure 2 can be seen. Security level annotations for input and output are added, as well as a declassification of `x` in the IF condition. Once the analysis is activated, illegal flows are highlighted in the source code. In the example, the illegal flow from the secret `inputPIN` via `x` via `y` to the public `print(y)` can be seen; due to the declassification, the flow to `print(0)`

[1] Integrity is dual to confidentiality, thus in the following we only discuss the latter. JOANA can handle both.
[2] `joana.ipd.kit.edu` provides download, webstart application, and other information
[3] In fact JOANA allows arbitrary lattices of security levels.

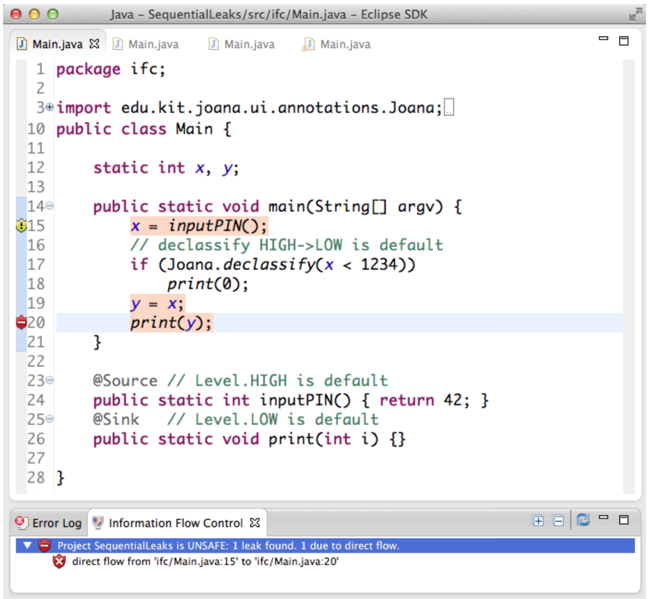


Figure 1: JOANA screenshot demonstrating classification, declassification, and illegal flow.

has been suppressed (these flows are explained in more detail in section 3). Full details on an illegal flow are available on demand. JOANA offers various options for analysis precision (e.g. object-sensitive points-to analysis, time-sensitive backward slicing). JOANA analyses Java bytecode and uses IBM's WALA analysis frontend; recently, a frontend for Android bytecode was added.

JOANA was able to provide security guarantees for several examples from the literature which are considered difficult, such as the EuroStoxx program in [20]. More interesting is perhaps the successful analysis of an experimental e-voting system developed by Küsters et al [18]. In a scalability study, the full source code of the HSQLDB database was analysed; analysis needed one day on a standard PC. Industrial applications are in preparation.

3 Probabilistic Noninterference

IFC for sequential programs must discover explicit and implicit leaks, which arise if (parts of) secret values are copied to public variables, resp. if secret values influence control flow (see example (1) in Figure 2). IFC for multi-threaded programs is much more challenging, as it must additionally prevent possibilistic or probabilistic information leaks. Both types of leaks depend on the interleaving of concurrent threads: possibilistic leaks may or may not occur depending on a specific interleaving, while probabilistic leaks exploit the probability distribution of interleaving orders. Example (2) in Figure 2 has a possibilistic leak, e.g., for interleaving order $5, 8, 9, 6$, which causes the secret PIN to be printed on public output. Example (3) has no possibilistic channel leaking PIN information. But the PIN's value may alter the probabilities of public outputs, because the running time of the loop may influence the interleaving order of the two assignments to x.

Most IFC approaches check some form of noninterference, and to this end classify program variables, input and output as high (secret) or low (public). Noninterference in its simplest form then demands that variations in secret input data do not cause variations in public output data ("low-equivalent inputs cause low-equivalent outputs") [23]. For concurrent programs with threads, *Probabilistic Noninterference* (PN) [27, 25, 24, 26, 19] is the established security criterion. PN explicitly allows nondeterminism in programs and demands that the probability of any observable behaviour is not influenced by secret values.[4] It is difficult to guarantee PN, as an IFC must in principle check all possible interleavings and their impact on execution probabilities.

One specific form of PN however is scheduler independent: *Low-Security Observational Determinism* (LSOD) demands that for a program which runs on two low-equivalent inputs, all possible traces are low-equivalent [22, 32, 14]. The following criterion is sufficient to guarantee LSOD and hence PN [32, 5]: 1. program parts contributing to low-observable behaviour are free of execution order conflicts, i.e. there is no low nondeterminism; 2. implicit or explicit flows do not leak high data to low-observable behaviour. Scheduler independence is a big practical advantage, but note that LSOD will absolutely prohibit any (even secure) low-nondeterminism.

4 IFC Algorithms

IFC algorithms check noninterference for a given program. For sequential programs, many IFC algorithms have been published, and several tools are available (see section 10). However not all algorithms can handle full sequential Java (or C), and some are rather imprecise and cause many false alarms. In particular, context-insensitive algorithms cause false alarms because they merge different calls to the same procedure (example (4) in Figure 2), and flow-insensitive algorithms cause false alarms because they ignore statement order (example (5) in Figure 2). For object-oriented programs, object- and field-sensitivity are similarly important [12]. Many security type systems used for IFC (such as [27] and its various successors) are however flow- and/or context-insensitive and will reject examples (4) and/or (5). The type system in [15] is flow-sensitive, but not context-sensitive.

For multi-threaded programs, various algorithms have been proposed which check probabilistic noninterfe-

4 Due to lack of space, we cannot explain the rather intricate technical definitions and issues of PN

(1)

```
void main():
  x = inputPIN();
  // inputPIN is
  // secret
  if (x < 1234)
     print(0);
  y = x;
  print(y);
  // public output
```

(2)

```
void main():
  fork thread_1();
  fork thread_2();
void thread_1():
  x = input();
  print(x);
void thread_2():
  y = inputPIN();
  x = y;
```

(3)

```
void main():
  fork thread_1();
  fork thread_2();
void thread_1():
  x = 0;
  print(x);
void thread_2():
  y = inputPIN();
  while (y != 0)
     y--;
  x = 1;
```

(4)

```
void main():
  h = inputPIN();
  l = 2;
  // l is public
  x = f(h);
  y = f(l);
  print(y);

int f(int x)
  {return x+42;}
```

(5)

```
void main():
  fork thread_1();
  fork thread_2();
void thread_1():
  l = 42;
  h = inputPIN();
void thread_2():
  print(l);
  l = h;
```

Figure 2: Small but typical leaks and precision problems. Programs 1 – 3 leak secret data to public output. (1) Explicit and implicit leaks, (2) possibilistic leak, (3) probabilistic leak. Programs 4 and 5 are secure, but only a precise analysis will see this. (4) context-insensitive analysis will generate a false alarm because calls to `f` are merged, (5) flow-insensitive analysis will generate false alarm because statement order in `thread_2` is ignored.

rence, e.g. [27, 25, 32, 14]. Unfortunately, most of these algorithms eventually turned out to be very restrictive, or had soundness problems. For example, [27] does not work with round-robin scheduling, [32] had a soundness leak, and [14] does not allow low statements after loops with high guards (and thus will reject almost any practical program).

IFC based on program dependence graphs (PDGs) was introduced because PDGs are naturally flow- and context-sensitive [12]. This not only improves precision for sequential IFC, but also avoids the above-mentioned problems with PN algorithms. Figure 3 presents two example PDGs. Nodes represent program statements or expressions, edges represent data dependencies, control dependencies, or inter-thread data dependencies. We will not discuss PDG details; it is sufficient to know the *Slicing Theorem*:

Theorem [13]. If there is no PDG path $a \to^* b$, it is guaranteed that there is no information flow from statement a to statement b in any program run.

Thus all statements which might influence a specific program point are those on backward paths from this point (the so-called “backward slice”). The PDG can be used to check whether there are any explicit or implicit leaks; technically it is required that no high source is in the backward slice of a low sink. This criterion is enough to guarantee sequential noninterference.

Note that the slicing theorem does not cover physical side channels such as power consumption profiles, nor does it cover corrupt schedulers or defect hardware; it only covers “genuine” program behaviour. There are stronger versions of the theorem, which consider only paths which can indeed be dynamically executed (“realizable” paths); these make a big difference in precision e.g. for programs with procedures or threads.

The construction of precise PDGs for full languages is absolutely nontrivial and requires additional information such as points-to analysis and exception analysis [12]; there is an abundant literature on PDG construction. C. Hammer was the first to introduce object- and field-sensitive PDGs for Java [10]. J. Krinke and D. Giffhorn provided precise algorithms for multi-threaded PDGs [5]. Today PDGs for full Java or C can handle several hundred thousand LOC.

5 The RLSOD Criterion

D. Giffhorn demonstrated that flow-sensitivity is the key to eliminate the above-mentioned problems with PN algorithms, and proposed a new noninterference criterion in the LSOD tradition. The criterion is called Relaxed Low-Security Observational Determinism (RLSOD) and is based on PDGs for multi-threaded programs. RLSOD is termination insensitive[5], but flow- context- and object-sensitive. From a practical viewpoint, we believe that these features are more important than termination sensitivity. RLSOD will in particular allow secure low-nondeterminism, while guaranteeing soundness.

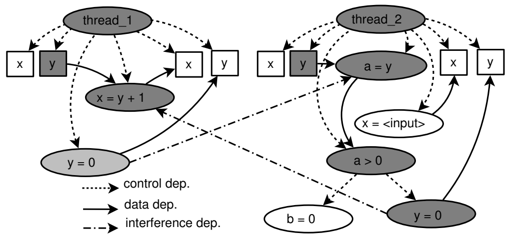
To understand RLSOD, consider Figure 3 (right), which presents the PDG for Figure 2 (5). `inputPIN` is annotated low, and `print` is annotated high. The security level of all other PDG nodes is computed by a fixpoint iteration [12]. RLSOD first checks for explicit or implicit leaks, exploiting the slicing theorem. There is no path from “`l=h;`” to “`print(l);`”; and hence no path from “`inputPIN()`” to “`print(l);`”. Therefore it is guaranteed that the printed value of `l` is not influenced by the secret `inputPIN`. More technically, statement “`l=h;`” is not considered low-observable and does not contribute to public behaviour; this is a consequence of the fact that in the PDG-based approach – due to flow-sensitivity – variables are *not* globally classified as secret or public (for details, see [6, 12]). Hence the example does not contain explicit or implicit leaks.

For probabilistic noninterference, according to LSOD one must additionally show that public output is not

[5] Due to lack of space, we cannot discuss the subtleties of termination sensitivity in connection with (R)LSOD; see [6, 5]

```
                        void thread_2():
int x, y;                 a = y;
                          x = <input>;
void thread_1():          if a > 0

  x = y + 1;                b = 0;
  y = 0;                  else
                            y = 0;
```



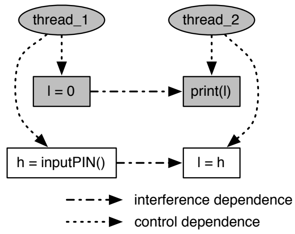


Figure 3: Left/middle: Example for a multi-threaded program and its PDG (from [6]). The backward slice of `y=0` is shaded. Right: PDG for program (5) in Figure 2. The backward slice of `print(l)` is shaded.

influenced by execution order conflicts such as data races [32, 6]. This can again be checked using PDGs and an additional analysis called "May happen in parallel" (MHP); the latter will uncover potential execution order conflicts or races. The complex technical details and proofs can be found in [6, 5]. In the example there are no possible execution order conflicts between secret operations which may influence the print statement. However statements `l=42;` and `print(l)` – which are both classified low – can be executed in parallel, and the scheduler nondeterministically decides which executes first – resulting in either `42` or `0` to be printed. Thus there is visible low nondeterminism, which is prohibited by classical LSOD. The program however is secure in the sense of PN (see [6] for a formal definition of PN and a detailed analysis of this example).

RLSOD allows secure low-nondeterminism. Technically, RLSOD allows execution order conflicts between low observable events, if these cannot be reached from high events. The latter condition can easily be checked in the multi-threaded control flow graph (CFG): if there are no paths in the CFG from a high statement to two different low statements which can be executed in parallel, no execution will ever transport secret information to the public nondeterminism. In the above example, the execution order conflict between `l=42;` and `print(l)` cannot be reached from high values. Hence the RLSOD criterion is fulfilled, and the program is guaranteed to be secure by JOANA. The example demonstrates that only the "R" optimization makes the LSOD idea practically usable.

6 Soundness Proof

Informally, the soundness property is stated as follows.

Theorem. If the RLSOD criterion is fulfilled for a program, it is probabilistically noninterferent. ☐

Soundness is based on a simpler theorem for sequential programs without threads:

Theorem. If no high data source is in the (context-sensitive) backward slice of a low data sink, a program is (sequentially) noninterferent. ☐

Snelting's original proof is in [28]. Later, Wasserrab provided a machine-checked proof which relies on a formal semantics for Java and PDGs [31, 30]. In [5, 6] Giffhorn showed the following much stronger theorem for multi-threaded programs and LSOD. It relies on discovering data conflicts (i.e. for two statements which can be executed in parallel, one writes a variable, and the other uses the same variable) and order conflicts (i.e. two low statements which can happen in parallel).

Theorem. If

1. no high source is in the (multi-threaded) backward slice of a low sink;
2. statements which can happen in parallel (MHP) are not both classified low;
3. for any low sink, no data conflict is in its backward slice

then LSOD and hence PN holds. The resulting PDG-based LSOD check is scheduler independent. ☐

For the "R"LSOD optimization, a machine-checked soundness proof is in progress. RLSOD is independent of scheduler policies. But note that a malicious scheduler could make the execution order of some low events dependent on high values, which under RLSOD might create a leak, as in `{h=0;||h=1;}; {l=0;||l=1;}`. The "R" optimization thus assumes, similar to [25], that the scheduler does not depend on high data.

7 Modular Analysis

PDGs essentially depend on a whole-program analysis which needs the complete source or byte code including libraries. In the scope of DFG priority program "Reliably Secure Software Systems" we however developed a modular variant of PDGs which allows to perform IFC analysis on isolated components and adjusts global PDG structure upon plug-in of a component.

PDGs for isolated components are based on *conditional dependencies* which are only valid if the plug-in context fulfills some additional conditions, written as component annotations. These conditions are boolean expressions over may-alias properties at the plug-in site (see example in Figure 4). Annotations induce a partial order on component contexts:

$C_1 \sqsubseteq C_2 \equiv v_1\ mayAlias\ v_2\ in\ C_1 \Rightarrow v_1\ mayAlias\ v_2\ in\ C_2$

The Monotonicity Property allows to guarantee IFC properties for components without reanalyzing them in a "smaller" context:

$$C_1 \sqsubseteq C_2 \wedge \textit{component noninterferent in } C_2 \Rightarrow \textit{component noninterferent in } C_1$$

```
class A { int i; }

@ifc: alias(a,b) => b -> c
int test(A a, A b):
  int c = a.i+42;
  return c;

@ifc: ? => h -!-> v2.i
void set(A v1, A v2, int h):
    v1.i = h;
```

```
void main():
  A v1 = new A();
  A v2 = new A();
  int h = inputPIN(); // secret
  set(v1, v2, h);
  print(v2.i);        // ok
  v2 = v1;
  set(v1, v2, h);
  print(v2.i);        // leak
```

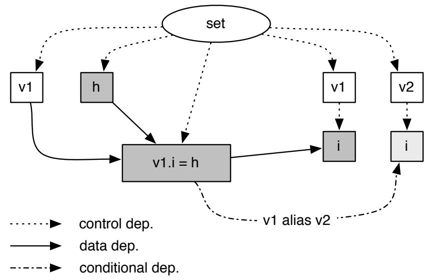


Figure 4: A small isolated component with annotations (left); corresponding modular PDG with conditional dependency (right). JOANA can infer a sufficient condition that guarantees no flow from `h` to `v2.i` through the annotation "`? => h -!-> v2.i`", yielding "?" $\equiv \neg alias(v1, v2)$.

The monotonicity property is also used to *infer* sufficient conditions on the context, which guarantee IFC properties for a component. In Figure 4, method `set()` may write the secret value of parameter `h` to `v2.i` depending on the context it is called in; JOANA infers that the component is safe if there is no may-alias between `v1` and `v2`. Details are described in [9].

8 Lock-sensitive Analysis

Recently, time-sensitive and lock-sensitive algorithms have been added to JOANA. Both are expensive, but can in difficult cases be activated on demand and eliminate more false alarms. Time sensitivity means that only PDG paths are considered, which can indeed be realized by a scheduler. Impossible execution orderings ("time travel") are excluded [6].

Lock sensitivity was investigated in a cooperation with M. Müller-Olm, because the original JOANA MHP analysis does not analyse explicit locks; it only analyses thread invocation structure (in a context-sensitive manner [5]). For a fine-grained analysis of locks, Müller-Olm's Dynamic Pushdown Networks [4] have been implemented for full Java and integrated into MHP analysis. Experiments demonstrated that indeed spurious dependencies between threads disappear, but the precision improvement is moderate [8].

9 Future Work

For a better effect of lock-sensitivity, a precise must-alias analysis for synchronization objects is necessary, which is however notoriously difficult for Java. Another possibility is to incorporate a technique called random isolation [16]. Improving the "R" optimization will enable even more low-nondeterminism.

A topic for future work is IFC for distributed systems with message passing. Such systems will be abundant e.g. in future energy or traffic systems. Message passing opens a new can of worms, because even if messages are encrypted, analysis of message patterns can, in combination with some brutal program analysis of the source code, recover secret program data. This interesting potential attack will be described in detail elsewhere.

10 Related Work

JIF [21] is one of the oldest IFC tools, but requires a special Java dialect and many annotations. The perhaps commercially most successful tool is TAJ / Andromeda by IBM [29]. The tool discovers only explicit leaks, but this restriction boosts scalability to millions of LOC. FlowDroid [1] does not handle probabilistic leaks, but offers good support for Android apps and dynamically configured systems. Some tools such as TaintDroid [3] perform a dynamic IFC at runtime. Dynamic IFC cannot give guarantees, but effectively finds many explicit leaks. Recently, combinations of static and dynamic analysis have been investigated, which seem promising for script languages [2].

11 Conclusion

Today, JOANA is one of the very few IFC tools worldwide which can handle full Java with unlimited threads, and – thanks to the underlying stack of sophisticated program analysis – offers good precision and scalability. We hope to be able to report industrial applications, as well as a full machine-checked proof for RLSOD, in the near future.

Acknowledgements. JOANA was supported by DFG (including DFG SPP 1496 "Reliably secure software systems") and BMBF in the scope of the software security competence center KASTEL.

Literature

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. PLDI*, page 29, 2014.

[2] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *Proc. Principles of Security and Trust POST*, pages 159–178, 2014.

[3] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM*, 57(3):99–106, 2014.

[4] Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *VMCAI*, pages 199–213, 2011.

[5] Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, May 2012.

[6] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, April 2014. To appear. Electronic preprint at `http://link.springer.com/article/10.1007%2Fs10207-014-0257-6`.

[7] Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs - a practical guide. In *Proc. 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2013.

[8] Jürgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive interference analysis for Java: Combining program dependence graphs with dynamic pushdown networks. In *Proc. 1st International Workshop on Interference and Dependence*, January 2013.

[9] Jürgen Graf. *Information Flow Control with System Dependence Graphs — Improving Modularity, Scalability and Precision for Object Oriented Languages.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2014. Forthcoming.

[10] Christian Hammer. *Information Flow Control for Java.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[11] Christian Hammer. Experiences with PDG-based IFC. In F. Massacci, D. Wallach, and N. Zannone, editors, *Proc. ESSoS'10*, volume 5965 of *LNCS*, pages 44–60. Springer-Verlag, February 2010.

[12] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.

[13] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proc. POPL '88*, pages 146–157, New York, NY, USA, 1988. ACM.

[14] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *Proc. 19th CSFW*, page 3. IEEE, 2006.

[15] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL '06*, pages 79–90. ACM, 2006.

[16] Nicholas Kidd, Thomas W. Reps, Julian Dolby, and Mandana Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, April 2011.

[17] Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. Extending and applying a framework for the cryptographic verification of java programs. In *Proc. Principles of Security and Trust (POST)*, pages 220–239. Springer, 2014.

[18] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of Java-like programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th.* IEEE Computer Society, June 2012.

[19] Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Proc. ESORICS*, volume 6345 of *LNCS*, pages 116–133, 2010.

[20] Heiko Mantel, Henning Sudbrock, and Tina Kraußer. Combining different proof techniques for verifying information flow security. In *Proc. LOPSTR*, volume 4407 of *LNCS*, pages 94–110, 2006.

[21] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[22] A. W. Roscoe, Jim Woodcock, and L. Wulf. Non-interference through determinism. In *ESORICS*, volume 875 of *LNCS*, pages 33–53, 1994.

[23] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[24] Andrei Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. 5th International Andrei Ershov Memorial Conference*, volume 2890 of *LNCS*, Akademgorodok, Novosibirsk, Russia, July 2003.

[25] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. CSFW '00*, page 200, Washington, DC, USA, 2000. IEEE Computer Society.

[26] Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[27] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. POPL '98*, pages 355–364. ACM, January 1998.

[28] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[29] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proc. PLDI*, pages 87–97, 2009.

[30] Daniel Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

[31] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-based noninterference and its modular proof. In *Proc. PLAS '09*. ACM, June 2009.

[32] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. CSFW*, pages 29–43. IEEE, 2003.

Authors During the last years, several people contributed to JOANA (from left to right). G. Snelting is a full professor for Informatics at KIT. His research group works on compilers, code optimization, program analysis, information flow control, and verification. Snelting originally invented PDG-based IFC and contributed to various JOANA aspects. Dennis Giffhorn developed the original RLSOD criterion and its soundness proof. Jürgen Graf developed the modular analysis. Christian Hammer developed the original JOANA kernel for Java. Martin Hecker extends JOANA for distributed systems. Martin Mohr works on lock-sensitive IFC analysis. Daniel Wasserrab provided Isabelle-checked soundness proofs for PDG-based IFC.

Address: Karlsruhe Institute of Technology, Faculty of Informatics, D-76131 Karlsruhe, E-Mail: gregor.snelting@kit.edu