Journal of Computer Security 0 (2018) 1–0
IOS Press

Low-Deterministic Security For Low-Nondeterministic Programs[1]

Simon Bischof, Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr and Gregor Snelting
Karlsruhe Institute of Technology, Germany
E-mail: gregor.snelting@kit.edu

Abstract.
We present a new algorithm, together with a full soundness proof, which guarantees *probabilistic noninterference* (PN) for concurrent programs. The algorithm follows the "low-deterministic security" (LSOD) approach, but for the first time allows general low-nondeterminism as long as PN is not violated.

The algorithm is based on the earlier observation by Giffhorn and Snelting that low-nondeterminism is secure as long as it is not influenced by high events [1]. It uses a new system of classification flow equations in multi-threaded programs, together with inter-thread / interprocedural dominators. Compared to LSOD and even [1], precision is boosted and false alarms are minimized. We explain details of the new algorithm and its soundness proof.

The algorithm is integrated into the JOANA software security tool, and can handle full Java with arbitrary threads. We apply JOANA to a multi-threaded e-voting system, and show how the algorithm eliminates false alarms. We thus demonstrate that low-deterministic security is a highly precise and practically mature software security analysis method.

Keywords: Software Security, Information Flow Control, Probabilistic Noninterference, Program Analysis

1. Introduction

Information flow control (IFC) analyses a program's source or byte code for security leaks, namely violations of confidentiality and integrity. IFC algorithms usually check some form of noninterference [3]. Sound IFC algorithms guarantee to find all leaks, while precise algorithms generate no false alarms. Unfortunately, perfect precision and soundness cannot be achieved together: the famous Rice theorem states that such perfect program analysis is undecideable. Thus many algorithms and definitional variations for noninterference have been proposed, which vary in precision, scalability, language restrictions, necessary annotations, and other factors.

Concurrent or multi-threaded programs introduce new threats to security, as nondeterminism and interleaving can create subtle leaks which are much more difficult to find or prevent than in sequential programs. For multi-threaded programs, *probabilistic noninterference* (PN) as introduced in [4–6] is the established security criterion. One of the oldest and simplest criteria which enforces PN is low-security observational determinism (LSOD), as introduced by Roscoe [7], and improved by Zdancewic, Huisman, and others [8, 9]. For LSOD, a relatively simple static check can be devised; furthermore LSOD is scheduler independent – which is a big advantage. However Huisman and other researchers found

[1] A preliminary version of parts of this work appeared in Proc. Principles of Security and Trust (POST '16) [2]. This work was partially supported by DFG grants Sn11/12-1/2/3 in the scope of the priority program "Reliably Secure Software Systems".

subtle soundness problems in earlier LSOD algorithms (which were mostly related to nonterminating programs), so Huisman concluded that scheduler-independent PN is not feasible [10]. Worse, LSOD strictly prohibits any, even secure low-nondeterminism – which makes LSOD unsuitable for practical purposes.

It is the aim of this article to demonstrate that improvements to LSOD can be devised, which invalidate these earlier objections. An important step was already provided by Giffhorn [1, 11] who discovered that

(1) an improved definition of low-equivalent traces can solve earlier soundness problems for infinite traces and nonterminating programs;
(2) flow- and context-sensitive program analysis is the key to a precise and sound LSOD algorithm;
(3) the latter can naturally be implemented through the use of program dependence graphs;
(4) additional support by may-happen-in-parallel analysis, precise points-to analysis and exception analysis makes LSOD work and scale for full Java;
(5) secure low-nondeterminism can be allowed by relaxing the strict LSOD criterion, while maintaining soundness.

Giffhorn's algorithm was the first to allow low nondeterminism, while basically maintaning the LSOD approach. The algorithm was described in detail in [1, 12]; it is integrated into the JOANA IFC tool.[2]

But Giffhorn's discovery was just a first step. In this paper, we describe new improvements for LSOD, which boost precision and reduce false alarms compared to original LSOD and even Giffhorn's algorithm. We first recapitulate technical properties of PN and LSOD. We then explain the new relaxed LSOD (RLSOD)[3] criterion in detail. It is based on the notion of dominance in threaded control flow graphs, and on fixpoint iteration in program dependence graphs.

The main contribution of this article, as compared to the preceding conference version [2], is a full soundness proof, which in turn led to an even more general formulation of the RLSOD criterion. RLSOD was recently integrated into JOANA. We present a case study, namely a prototypical e-voting system with multiple threads, as well as performance and scalability measurements.

Our work builds heavily on our earlier contributions [1, 12], but the current article is aimed to be self-contained. We begin with an overview of the RLSOD framework and its attacker model.

2. The RLSOD framework and its assumptions

2.1. Language Assumptions

The RLSOD approach is based on Program Dependence Graphs (PDGs, see section 4.1 for details) and thus can be used with any imperative or object-oriented language for which a PDG can be constructed. PDGs are naturally flow- and context-sensitive, thus improving precision (see below). PDG construction must be sound and will then fulfill the assumptions of the slicing theorem (sec. 4.1); sound PDGs have been constructed for C, full Java (see [12]), and many other languages. Thus there are no restrictions on language or control flow. In the current paper, we assume a simple imperative language with procedures

[2] see `joana.ipd.kit.edu`. JOANA can analyse full Java with arbitrary threads, and was applied in various projects [13–16]. Usage of JOANA is described in [17].

[3] Giffhorn's original criterion was called RLSOD in [1, 2, 18], and the new criterion was called iRLSOD in [2]. In this article, the iRLSOD criterion and its improvements are called RLSOD, while Giffhorn's original criterion is called "Giffhorn's algorithm".

```
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("J");
7  void thread_2():
8    read(H);
9    while (H != 0)
10      H--;
11   print("CS");
```

Fig. 1. Some leaks. Left: explicit and implicit, middle: possibilistic, right: probabilistic. For simplicity, we assume that `read(L)` reads low variable `L` from a low input channel; `print(H)` prints high variable `H` to a high output channel. Note that reads of high variables are classified high, and prints of low variables are classified low.

and threads; a formal PDG construction for this language, together with a machine-checked soundness proof was provided in [19]. We would however like to point out some additional assumptions for multi-threaded programs.

We assume that statements resp. byte codes are deterministic, and that non-determinism can only arise from scheduling choices. In particular, for a given statement, results and chosen branch (for conditionals) only depend on values read by the statement. The program can read input via input statements and produce output via output statements. At the beginning of execution, the *START* statement is the only statement that can be scheduled.[4] After executing a statement, its dynamic control flow successors can be scheduled (i.e. for conditionals, the first statement from the chosen branch; for forks the intra-thread successor and the first statement of the forked thread).

We therefore assume – like most other authors, e.g. [5, 20] – *interleaving semantics*. Under the Java Memory Model, the compiler may, e.g. through reordering, in rare cases produce code which is not consistent with interleaving semantics. Allowing such behavior limits static analysis considerably, so most authors ignore this possibility.

We also assume that the program always terminates. Possibly non-terminating programs, e.g. truly interactive programs, lead to the problem of termination leaks [21]. Subtleties regarding nontermination are discussed in section 4.2. Since we assume terminating programs, we can ignore those problems.

2.2. Scheduler Assumptions

RLSOD requires that the scheduler is truly probabilistic. This means that for two statements c_1 and c_2 that can be scheduled, the relative probability to be scheduled next is independent of other possibly running threads, current program state, or the execution up to this point. It might however depend on the statements c_1 and c_2 themselves.[5] The necessity of this assumption was stressed by various authors, e.g. [5, 22]. To illustrate it, we assume a standard *low*/*high* classification for program variables (see section 3 for details). A malicious scheduler can then easily read high values to construct an explicit flow by scheduling, as in

[4] JOANA can also handle the Android Life cycle, see [13].

[5] Note that this is less restrictive than a uniform scheduler, where that relative probability is always 1. A truly probabilistic scheduler on the other hand can assign priorities to statements, e.g. increase the priorities for all statements of a specific (syntactic) thread.

```
1  void thread1():
2    if (H) {
3       skip;};
4    fork thread2();
5    print(17);
6
7  void thread2():
8    print(42);
```

Fig. 2. Deterministic round-robin scheduling may leak.

`{H=0;||H=1;} {L=0;||L=1;}` :
the scheduler can leak `H` by scheduling the `L` assignments after reading `H`, such that the first visible `L` assignment represents `H`.

Even if the scheduler is not malicious, but follows a deterministic strategy which is known to the attacker, leaks can result. As an example, consider Figure 2. Assume deterministic round robin scheduling which executes 3 basic statements per time slice. Then for `H=1` statements 2,3,4,9,5 are executed, while for `H=0`, statements 2,4,5,9 are executed. Thus the attacker can observe the public event sequence 9→5 resp. 5→9, leaking `H`. However under the assumption of truly probabilistic scheduling, Figure 2 is probabilistic noninterferent.

Thus allowing round robin schedulers severely restricts secure programs. This phenomenon has been observed by earlier authors (see section 9 for details). RLSOD accepts the example in Figure 2 as secure and therefore is not sound when used with round robin. Probabilistic scheduling is the price to pay for RLSOD's much better precision and freedom from program restrictions. We believe that in practice, scheduler restrictions are more acceptable than program restrictions or false alarms.

2.3. Attacker Model

The traditional sequential attacker model assumes that the attacker can see the low part of both the initial state and the final state. Additionally, a static classification of all program variables is assumed. This means the attacker can see one part of the memory, but cannot see the other. For programming languages like C or Java, which include local variables and thus use an activation record stack, this implies that the memory addresses the attacker can observe change during execution. We therefore believe that this attacker model is unrealistic. Thus we assume that the attacker cannot see any internal memory of the program, and instead can only see low external input or low output events. The engineer has to provide information about the set of input and output statements and their classification. We further assume that the attacker might see multiple runs with the same input. For multi-threaded (nondeterministic) programs, where the probability of attacker-observable behaviors depends on the secret, the attacker might infer something about the secret by counting the different low-observable behaviors. This motivates the use of probabilistic noninterference.

2.4. The role of flow- and context-sensitivity

PDGs have been chosen as the basis for JOANA and RLSOD because PDGs are automatically flow- and context-sensitive. That is, they respect statement order, and respect call sites of procedures. In contrast, most security type systems are flow insensitive, and even the flow-sensitive type system in [23] is not context-sensitive. Thus statement order is ignored, and all call sites of a procedure are merged; this

heavily reduces precision and increases false alarms (see [12, 24] for a more detailed discussion; section 9 presents additional examples). In the current paper, all definitions and proofs assume flow-sensitivity; this explains why some definitions differ from their "classical" version.

But note that there is a price to pay for flow- and context-sensitivity: RLSOD is not compositional. That is, RLSOD always needs the complete source or byte code. In practice however, stubs for missing functions can be used, which "simulates" compositionality (see case study in section 7). Compositionality is discussed in more detail in section 9.

3. Noninterference and LSOD

IFC guarantees that no violations of confidentiality or integrity may occur. For confidentiality, program variables are classified as "high" (H, secret) or "low" (L, public), and it is assumed that an attacker can see low values, but cannot see any high value.[6]

Figure 1 presents small but typical confidentiality leaks. As usual, variable `H` is "High" (secret), `L` is "Low" (public). Explicit leaks arise if (parts of) high values are copied (indirectly) to low output. Implicit leaks arise if a high value can change control flow, which can change low behaviour (see Figure 1 left). Possibilistic leaks in concurrent programs arise if a certain interleaving produces an explicit or implicit leak; in Fig. 1 middle, interleaving order 5, 8, 9, 6 causes an explicit leak. Probabilistic leaks arise if the probability of low output is influenced by high values; in Fig. 1 right, `H` is never copied to `L`, but if the value of `H` is large, probability is higher that "`JCS`" is printed instead of "`CSJ`". Thus when the program is run multiple times, by observing the distribution of results, the attacker can obtain information about the secret, namely estimations of the value of `H`.

3.1. Sequential Noninterference

Before we formalize RLSOD, let us repeat the classical definition of sequential noninterference. The classic definition assumes that a global and static classification $cl(v)$ of all program variables v as secret (H) or public (L) is given. It therefore assumes that the whole memory is divided into a part the attacker can observe and a part hidden from the attacker. Note that flow-sensitive IFC such as RLSOD does *not* use a static, global classification of variables; this will be explained in section 4.1 (Definition 9).

Definition 1 (Sequential noninterference)**.** *Let $\mathcal{P}$ be a program. Let s, s' be initial program states, let $[\![\mathcal{P}]\!](s)$, $[\![\mathcal{P}]\!](s')$ be the final states after executing $\mathcal{P}$ in state s resp. s'. Noninterference holds iff*

$$s \sim_L s' \implies [\![\mathcal{P}]\!](s) \sim_L [\![\mathcal{P}]\!](s').$$

The relation $s \sim_L s'$ means that two states are low-equivalent, that is, coincide on low variables: $\forall v\colon cl(v) = L \implies s(v) = s'(v)$. Classically, program input is assumed to be part of the initial states s, s', and program output is assumed to be part of the final states; the definition can be generalized to work with explicit input and output streams.

[6] A more detailed discussion of IFC attacker models can be found in e.g. [1]. Note that JOANA allows arbitrary lattices of security classifications, not just the simple $\bot = L \leqslant H = \top$ lattice. Note also that integrity is dual to confidentiality, but will not be discussed here. JOANA can handle both.

3.2. Probabilistic Noninterference

In multi-threaded programs, fine-grained interleaving effects must be accounted for, thus traces are used instead of states. Through loops or recursion, a statement might be executed multiple times within a trace. To distinguish those, we use the concept of operations as in [11]. A trace is a sequence of events $t = (s_1, o_1, s_2), (s_2, o_2, s_3), \ldots, (s_\nu, o_\nu, s_{\nu+1}), \ldots$, where the o_ν are operations (i.e. dynamically executed program statements c_ν; we write $stmt(o_\nu) = c_\nu$). Operations are unique within a trace. $s_\nu, s_{\nu+1}$ are the states before resp. after executing o_ν. We write ε for the empty trace. Since we assume programs to be terminating, all traces are finite.

For PN, the notion of low-equivalent traces is essential. Classically, traces are low equivalent if for every $(s_\nu, o_\nu, s_{\nu+1}) \in t$, $(s'_\nu, o_\nu, s'_{\nu+1}) \in t'$, it holds that $s_\nu \sim_L s'_\nu$ and $s_{\nu+1} \sim_L s'_{\nu+1}$. This definition enforces a rather restrictive lock-step execution of both traces. Later definitions (e.g. [5]) use stutter equivalence instead of lock-step equivalence; thus allowing one execution to run faster than the other ("stuttering" means that one trace performs additional operations which do not affect public behaviour). To formalize PN, we begin with

Definition 2. *N denotes the set of all program statements $\in \mathcal{P}$, $I \subseteq N$ the input statements ("sources"), $O \subseteq N$ the output statements ("sinks"), $ucl : I \cup O \to \{L, H\}$ the classification of sources and sinks as provided by the user ("engineer").*

The following definition of low-equivalent traces assumes flow-sensitivity. It is slightly more complex than the "classical" definition, but increases precision as explained below.

Definition 3. *(1) For an operation o, $def(o), use(o)$ are the program variables defined (i.e. assigned) resp. used in o. For state s and $d \subseteq dom(s)$, $s|_d$ is the projection of s onto d.*

(2) The low-observable part of an event is defined as

$$E_L((s, o, s')) = \begin{cases} (s|_{use(o)}, o, s'|_{def(o)}), & \text{if } ucl(o) = L \\ \varepsilon, & \text{otherwise} \end{cases}$$

(3) The low-observable subtrace of trace t is

$$LS(t) = map(E_L)(filter(\lambda e.E_L(e) \neq \varepsilon)(t)).$$

(4) Traces t, t' are low-equivalent, written $t \sim_L t'$, if $LS(t) = LS(t')$. Obviously, $\sim_L$ is an equivalence relation. Thus the low-class of t is

$$[t]_L = \{t' \mid t' \sim_L t\}.$$

Note that the $t' \in [t]_L$ cannot be distinguished by an attacker, as all $t' \in [t]_L$ have the same low behaviour. Thus $[t]_L$ represents t's low behaviour. Note also that the flow-sensitive projections $s|_{def}(o)$, $s|_{use}(o)$ are usually much smaller than a flow-insensitive, statically defined low part of s. This results in more traces to be low-equivalent without compromising soundness. This subtle observation is another reason why flow-sensitive IFC is more precise (cmp. [1], sec. 3).

PN is called "probabilistic", because it essentially depends on the probabilities for certain traces under certain inputs. Thus we define

Definition 4. *(1) $P_i(t)$ is the probability that a specific trace t is executed under input i.*
(2) $P_i([t]_L)$ is the probability that some trace $t' \in [t]_L$ (i.e. $t' \sim_L t$) is executed under i.

As $[t]_L$ is recursively enumerable, P_i is a discrete probability distribution, hence $P_i([t]_L) = \sum_{t' \in [t]_L} P_i(t')$.

The following PN definition uses explicit input streams instead of initial states. For all inputs the same initial state is assumed, but it is assumed that all inputs are classified low or high. Traditionally, input streams $i = i_1 i_2 \ldots$, $i' = i'_1 i'_2 \ldots$ are low equivalent ($i \sim_L i'$) if they coincide on low values: $cl(i_\nu) = L \wedge cl(i'_\nu) = L \implies i_\nu = i'_\nu$. But this definition is not realistic: Suppose there is a race between a high and a low input statement. Then the classification of the input depends on the race and is no longer independent of a specific execution. This makes it difficult to properly define low-equivalence of inputs. Consequently, Giffhorn used a different definition of low-equivalent inputs: instead of one input stream with input values of varying classification, Giffhorn assumes several input streams with fixed classification each. This leads to

Definition 5. *(1) An input i of a program consists of one input stream per security level; we write i_l for the input stream of security level l. An input statement $n \in I$ reads (and removes) the first value from $i_{ucl(n)}$.*
(2) Inputs are considered low equivalent if their low input streams are equal: $i \sim_L i' \iff i_L = i'_L$
(3) $T(i)$ is the set of all possible traces of a program $\mathcal{P}$ under input i.

Definition 6 (Probabilistic noninterference)**.** *Let i, i' be inputs; let $\Theta = T(i) \cup T(i')$. PN holds iff*

$$i \sim_L i' \implies \forall t \in \Theta \colon P_i([t]_L) = P_{i'}([t]_L)$$

That is, if we take any trace t which can be produced by i or i', the probability that a $t' \in [t]_L$ is executed is the same under i resp. i'. In other words, *probability for any public behaviour is independent from the choice of i or i'* and thus cannot be influenced by secret input.

If $t \notin T(i)$, $P_i(t) = 0$. Using the above sum property of $P_i([t]_L)$, the PN condition is thus equivalent to

$$i \sim_L i' \implies \forall t \colon \sum_{t' \in [t]_L} P_i(t') = \sum_{t' \in [t]_L} P_{i'}(t')$$

Applying this to Figure 1 right, we first observe that all inputs are low equivalent as there is only high input. For any trace t there are only two possibilities: `...print("J")...print("CS")...` $\in t$, or `...print("CS")...print("J")...` $\in t$. There are no other low events, hence there are only two equivalence classes

$$[t]^1_L = \{t' \mid \ldots\texttt{print("J")}\ldots\texttt{print("CS")}\ldots \in t'\}$$
$$[t]^2_L = \{t' \mid \ldots\texttt{print("CS")}\ldots\texttt{print("J")}\ldots \in t'\}$$

Now if i contains a small value, i' a large value, as discussed earlier $P_i([t]^1_L) \neq P_{i'}([t]^1_L)$ as well as $P_i([t]^2_L) \neq P_{i'}([t]^2_L)$, hence PN is violated.

In practice, the $P_i([t]_L)$ are difficult or impossible to determine. So far, only simple Markov chains have been used to explicitly determine the P_i for very small programs; where the Markow chain models the probabilistic state transitions of a program, perhaps together with a specific scheduler [5, 25]. Fortunately, explicit probabilities are not needed for our soundness proofs. As a sanity check, we demonstrate

that for sequential programs, PN implies sequential noninterference. Note that for sequential (deterministic) programs $|T(i)| = 1$, and for the unique $t \in_1 T(i)$ we have $P_i(t) = 1$.

Lemma 1. *For sequential programs, probabilistic noninterference implies sequential noninterference.*

Proof. Let $s \sim_L s'$. For sequential NI, input is part of the initial states, thus $i \sim_L i'$. Now let $t \in T(i), t' \in T(i')$, hence $t, t' \in \Theta$. Due to PN, $P_i([t]_L) = P_{i'}([t]_L)$ and $P_i([t']_L) = P_{i'}([t']_L)$. Due to sequentiality, $P_i([t]_L) = P_i(t) = 1$ and $P_{i'}([t']_L) = P_{i'}(t') = 1$. Hence $P_{i'}([t]_L) = P_{i'}([t']_L) = 1$. That is, with probability 1 the trace t' executed under i' is low equivalent to t. Thus in particular the final states in t resp. t' must be low equivalent. Hence $s \sim_L s'$ implies $[\![\mathcal{P}]\!](s) \sim_L [\![\mathcal{P}]\!](s')$. □

3.3. Low-deterministic Security

LSOD is the oldest and simplest criterion which enforces PN. LSOD demands that low-equivalent inputs produce low-equivalent traces. LSOD is scheduler independent and implies PN (see below). It is intuitively secure: changes in high input can never change low behaviour, because low behaviour is enforced to be deterministic. This is however a very restrictive requirement and eventually led to popular scepticism against LSOD.

Definition 7 (Low-security observational determinism)**.** *Let i, i', Θ as above. LSOD holds iff*

$$i \sim_L i' \implies \forall t, t' \in \Theta : t \sim_L t'.$$

Under LSOD, all traces t for input i are low-equivalent: $\forall t' \in T(i) : t' \sim_L t$, thus $T(i) \subseteq [t]_L$. If there is more than one trace for i, then this must result from high-nondeterminism; low behaviour is strictly deterministic.

Lemma 2. *LSOD implies PN.*

Proof. Let $i \sim_L i', t \in \Theta$. WloG let $t \in T(i)$. Due to LSOD, we have $T(i) \subseteq [t]_L$. As $P_i(t') = 0$ for $t' \notin T(i)$, we have $P_i([t]_L) = \sum_{t' \in [t]_L} P_i(t') = \sum_{t' \in T(i)} P_i(t') = 1$, and likewise $P_{i'}([t]_L) = 1$, so $P_i([t]_L) = P_{i'}([t]_L)$. □

Zdancewic [8] proposed the first IFC analysis which checks LSOD. His conditions require that

(1) there are no explicit or implicit leaks,
(2) no low observable operation is influenced by a data race,
(3) no two low observable operations can happen in parallel.

The last condition imposes the infamous LSOD restriction, because it explicitly disallows that a scheduler produces various interleavings which switch the order of two low statements which may happen in parallel, and thus would generate low nondeterminism. Besides that, the conditions can be checked by a static program analysis; Zdancewic used a security type system.

As an example, consider Figure 4. In Figure 4 middle, statements `print(L)` and `L=42` – 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 definitely secure according to PN, because the high read can only happen after the race outcome is already decided.

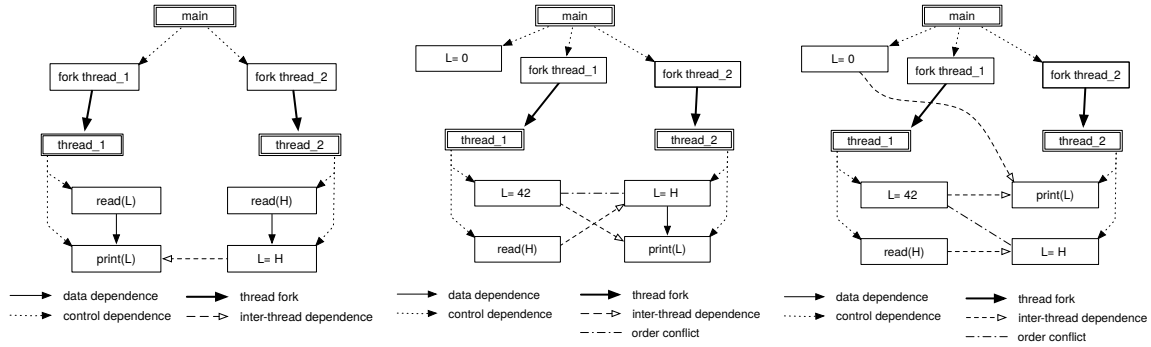


Fig. 3. Left to right: PDGs for Figure 1 middle, and for Figure 4 left and middle.

4. Giffhorn's Criterion

In this section, we recapitulate PDGs, their application for LSOD, and Giffhorn's original criterion. This discussion is necessary in order to understand the new improvements for RLSOD.

4.1. PDGs for IFC

Snelting proposed to use Program Dependence Graphs (PDGs) as a device to check integrity of software as early as 1995 [26]. Later the approach was expanded into the JOANA IFC project. It was shown that PDGs guarantee sequential noninterference [27], and that they provide improved precision as they are naturally flow- and context-sensitive [12].

In this paper, we just present three PDG examples and some explanations. PDG nodes represent program statements or expressions; edges represent data dependencies, control dependencies, inter-thread data dependencies, or summary dependencies. Figure 3 presents the PDGs for Figure 1 middle, and for Figure 4 left and middle. The construction of precise PDGs for full languages is absolutely nontrivial and requires additional information such as points-to analysis, exception analysis, and thread invocation analysis [12]. We will not discuss PDG details; it is sufficient to know the *Slicing Theorem* for sequential programs:

Theorem 1. *If there is no PDG path $a \rightarrow^* b$, it is guaranteed that statement a can never influence statement b. In particular, values computed in a cannot influence values computed in b.*

Proof. see [28] □

Thus all statements which might influence a specific program point b are those on backward paths from this point, the so-called "backward slice" $BS(b)$. In particular, information flow $a \rightarrow^* b$ is only possible if $a \in BS(b)$. 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, objects, or threads [12, 29, 30].

As an example, consider Figure 3. The left PDG has a data dependency edge from `L=H;` to `print(L);`, because `L` is defined in line 9 (Figure 4 left), used in line 10, there is a path in the control flow graph (CFG) from 9 to 10, and `L` is not reassigned ("killed") on the path. Thus there is a PDG path from `read(H);` to `print(L);`, representing an illegal flow from line 7 to line 10 (a simple explicit leak). In Figure 3 right, there is no path from `L=H;` to `print(L);` due to flow sensitivity: no scheduler

will ever execute `L=H;` before `print(L);`. Hence no path from `read(H)` to `print(L);` exists, and it is guaranteed that the printed value of `L` is not influenced by the secret `H`.

In general, the multi-threaded 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 (see theorem 2). For probabilistic noninterference, according to the Zdancewic LSOD criterion one must additionally show that public output is not influenced by execution order conflicts such as data races, and that there is no low nondeterminism. 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. Several precise and sound MHP algorithms for full Java are available today (see e.g. [11, 31, 32]). Note that an imprecise MHP analysis will substantially degrade the precision of (R)LSOD, and cause many false alarms (see [1, 11] for details).

In the following, we will need some definitions related to PDGs. For more details on PDGs, MHP, flow- context-, object- and time-sensitivity, see [12].

Definition 8. *Let $G = (N, \rightarrow)$ be a PDG, where N consists of program statements and expressions, and $\rightarrow$ comprises data dependencies, control dependencies, summary dependencies, and inter-thread dependencies. The (context-sensitive) backward slice for $n \in N$ is defined as*

$$BS(n) = \{m \mid m \rightarrow_R^* n\}$$

where $\rightarrow_R^*$ includes only realizable *(i.e. context-, object- and optionally time-sensitive) paths in the PDG.*

For PDG-based analyses, the inputs ("sources") and outputs ("sinks") are nodes in the PDG, so $I, O \subseteq N$. The soundness theorem for PDG-based sequential IFC can now be formalized:

Theorem 2. *Sequential noninterference holds if*
$\forall n \in I, n' \in O : ucl(n') = L \wedge ucl(n) = H \implies n \notin BS(n')$

Proof. Snelting's original proof was in [27]; additional details are given in [12]. □

The user ("engineer") classifications *ucl* in the PDG can be propagated to yield a classification *cl* of all statements, which provides an alternate way of checking sequential noninterference. This definition will be expanded later to cover concurrent programs.

Definition 9. *(1) The classification cl can be computed from ucl via the flow equation*

$$cl(n) = \bigsqcup_{m \rightarrow n} cl(m)$$

with additional constraints $\forall n \in I : ucl(n) \leqslant cl(n)$ and $\forall n \in O : ucl(n) \geqslant cl(n)$.
(2) For an operation o in a trace t, we have $stmt(o) \in N$ and define $cl(o) = cl(stmt(o))$.

Concerning *cl* it is important to note that PDGs are automatically flow- and context-sensitive, and may contain a program variable *v* several times as a PDG node; each occurence of *v* in *N* may have a different classification! Thus there is no global classification of variables, but only the local classification $ucl(n)$ together with the global flow constraints $cl(n) = \bigsqcup_{m \rightarrow n} cl(m)$. The latter can easily be computed by

```
1  void main():
2    L = 0;
3    fork thread_1();
4    fork thread_2();
5  void thread_1():
6    L = 42;
7    read(H);
8  void thread_2():
9    L = H;
10   print(L);
```

```
1  void main():
2    L = 0;
3    fork thread_1();
4    fork thread_2();
5  void thread_1():
6    L = 42;
7    read(H);
8  void thread_2():
9    print(L);
10   L = H;
```

```
1  void main():
2    L = 0;
3    read(H);
4    while (H2>0)
5      {H2--;}
6    fork thread_1();
7    fork thread_2();
8  void thread_1():
9    L = 42;
10   read(H);
11 void thread_2():
12   print(L);
13   L = H;
```

Fig. 4. Left: insecure program, obvious explicit leak. Middle: secure program, Giffhorn's criterion + flow sensitivity avoid false alarm. Right: only RLSOD avoids false alarm.

a fixpoint iteration on the PDG; which is initialized with the *ucl* values for source nodes (see [12]). If fixpoint iteration eventually computes $cl(n) > ucl(n)$ for a sink n, an explicit or implicit leak has been discovered [12]. JOANA offers additional support to analyse and localize leaks [17, 33].

4.2. LSOD with PDGs

In his 2012 thesis, Giffhorn applied PDGs to PN. He showed that PDGs can naturally be used to check Zdancewic's LSOD criteria, and provided a soundness proof as well as an implementation for JOANA [11]. Giffhorn also found the first optimization relaxing LSOD's strict low-determinism.

Giffhorn's motivation was to repair soundness leaks which had been uncovered in some previous LSOD algorithms. In particular, treatment of nontermination without being overly restrictive or allowing implicit leaks had proven to be tricky. Giffhorn provided a new definition for low-equivalent traces:

Definition 10. *Let t, t' be two finite or infinite traces. $t \sim_L t'$ iff*

(1) if t, t' are both finite, as usual the low events and low memory parts must coincide (see Definition 3);

(2) if wloG t is finite, t' is infinite, then this coincidence must hold up to the length of the shorter trace, and the missing operations in t must be missing due to an infinite loop *(and nothing else);*

(3) for two infinite traces, this coincidence must hold for all low events, or if low events are missing in one trace, they must be missing due to an infinite loop.

The formal version of this definition can be found in [1]. It turned out that conditions 2. and 3. not only avoid previous soundness leaks, but can precisely be characterized by dynamic control dependencies in traces [1]. Furthermore, the latter can soundly and precisely be statically approximated through PDGs (which include all control dependencies). Moreover, the static conditions identified by Zdancewic which guarantee LSOD can naturally be checked by PDGs, and enjoy increased precision due to flow-, context- and object-sensitivity.

In this paper however, we ignore the issue of termination completely and concentrate on the formalization and improvement of Giffhorn's LSOD check. We begin with some definitions.

Definition 11. *(1) We write $MHP(n,m)$ if MHP analysis concludes that n and m may be executed in parallel. Formally, $MHP(n,m)$ holds if traces t, t' exist where*
$t = \ldots(s_\nu, o_\nu, s_{\nu+1})\ldots(s_\mu, o_\mu, s_{\mu+1})\ldots$, $t' = \ldots(s'_\mu, o_\mu, s'_{\mu+1})\ldots(s'_\nu, o_\nu, s'_{\nu+1})\ldots$,
$n = stmt(o_\nu)$, $m = stmt(o_\mu)$.

(2) $lnd(n,n') \iff MHP(n,n') \wedge ucl(n) = ucl(n') = L$, which denotes that n, n' are low-nondeterministic;

(3) $race(n,n') \iff MHP(n,n') \wedge \exists v \in \big(def(n) \cap (def(n') \cup use(n'))\big)$, which denotes there is a data race between n, n' which can influence the value read at n'.

(4) $path(n,n') \iff n \rightarrow^*_{CFG} n'$ is a path in the CFG. We will also use $path(n,n')$ to denote the set of all nodes n'' on CFG paths from n to n'.

Then the PDG-based LSOD criterion – a formalization of Zdancewic's criterion using PDGs – requires:

Definition 12 (Giffhorn's criterion)**.**

1. $\forall n \in O, n' \in I\colon\ ucl(n) = L \wedge ucl(n') = H \implies n' \notin BS(n)$,
2. $\forall n, n' \in N, n'' \in O\colon\ race(n,n') \wedge ucl(n'') = L \implies n' \notin BS(n'')$,
3. $\forall n, n' \in I \cup O\colon \neg lnd(n,n')$.

Theorem 3. *Giffhorn's criterion implies LSOD.*

Proof. For proof and implementation details, see [1]. □

Condition 1 is just sequential noninterference (no explicit/implicit leaks, see theorem 2), condition 2 guarantees that no race is in the backward slice of a low sink, and condition 3 prohibits any low nondeterminism. Note that the race definition[7] from definition 11 is asymmetric: Giffhorn discovered that only two of the three classical race situations ("write-write, write-read, read-write") are relevant. The case $v \in def(n) \cap use(n'), n \in BS(n'')$ can never cause a leak because the value written at n is independent of the outcome of that race! Thus the example from the top of Figure 5 is considered secure, whereas a symmetric race definition would have caused a false alarm. Also note that conditions 2 and 3 are not completely disjoint, in particular if $ucl(n) = ucl(n') = ucl(n'') = L, n \in BS(n'')$ both conditions are violated.

Applying Giffhorn's criterion to Figure 1 right, it discovers a leak according to condition 3, namely low nondeterminism between lines 6 and 11; which is correct. For the example in Figure 5 bottom, condition 3 is not violated, but condition 2 is violated. In Figure 4 left, a leak is discovered according to condition 1, which is also correct (cmp. PDG example above). In Figure 4 middle and right, the explicit leak has disappeared (thanks to flow-sensitivity), but another leak is discovered by condition 2: we have $race(\texttt{L = 42;}, \texttt{print(L);})$ and $\texttt{print(L);} \in BS(\texttt{print(L);})$, which causes a false alarm.

The example motivates Giffhorn's optimized criterion: *low nondeterminism may be allowed, if it cannot be reached from high events*. That is, there must **not** be a path in the control flow graph from some n'', where $ucl(n'') = H$, to n or n', where $lnd(n,n')$. If there is no path from a high event to the low nondeterminism, no high statement can ever be executed before the nondeterministic low statements.

[7] In [11] these races were called "data conflicts".

```
1  void thread1():
2     doSomething();
3     L = 1;
4     X = 3*L;
5     print(X);
```

```
1  void thread_2():
2     H = inputPIN();
3     while (H>0) H--;
4     H = H+L;
```

```
1  void thread1():
2     H = 1;
3     doHigh();
4     H = 42;
```

```
1  void thread_2():
2     H = inputPIN();
3     while (H>L) H--;
4     if (H>41)
5        print(L);
```

Fig. 5. Races with one node in the backward slice of a low sink. Top: $n =$ "`L = 1`", $n' =$ "`H = H + L`", $n'' =$ "`print(X)`", $n \in BS(n'')$. Bottom: $n =$ "`H = 42`", $n' =$ "`H - -`", $n'' =$ "`print(L)`", $n' \in BS(n'')$.

Thus the latter can never produce visible behaviour which is influenced by high values. This argument leads to Giffhorn's optimized criterion, which replaces condition 3 from definition 12 by

3'. $\forall n, n' \in I \cup O\colon \big(\exists n'' \in I\colon ucl(n'') = H \wedge (path(n'', n) \vee path(n'', n'))\big) \implies \neg lnd(n, n')$

This condition can be rewritten by contraposition to the more practical form

3'. $\forall n, n' \in I \cup O\colon lnd(n, n') \implies \forall n'' \in (path(START, n) \cup path(START, n')) \cap I\colon ucl(n'') = L$

In fact Giffhorn's optimization works for data races as well: no data race may be in the backward slice of a low sink, *unless it is unreachable by high events*. That is, condition 2 can be improved the same way as condition 3, leading to

2'. $\forall n, n' \in N, n'' \in O\colon race(n, n') \wedge cl(n'') = L \wedge \exists n''' \in I\colon ucl(n''') = H \wedge \big(path(n''', n) \vee path(n''', n')\big)$
$\implies n, n' \notin BS(n'')$.

By contraposition, we obtain the more practical form

2'. $\forall n, n' \in N, n'' \in O\colon race(n, n') \wedge ucl(n'') = L \wedge n' \in BS(n'')$
$\implies \forall n''' \in (path(START, n) \cup path(START, n')) \cap I\colon ucl(n''') = L$

Remember that condition 2' is not symmetrical in n, n', due to Giffhorn's simplified race definition. Figure 1 right violates Giffhorn's criterion, because one of the low-nondeterministic statements, namely line 11, can be reached from the high statement in line 8; thus criterion 3' is violated. Indeed the example contains a probabilistic leak. Figure 4 middle is secure according to Giffhorn, because the data race between line 6 resp. 9 can not be reached from any high statement – condition 2 is violated (note that in this example, $n' = n''$), but 2' holds. Indeed the program is PN. Figure 4 right is however not covered by Giffhorn's criterion, because the initial `read(H2)` will reach any other statement. But the program is PN, because `H2` does not influence the data race determining the low behavior!

The example shows that Giffhorn's optimization does indeed reduce false alarms, but it removes only false alarms on low paths beginning at program start. Anything after the first high statement will usually be reachable from that statement, and does not profit from rule 3' resp. 2'. Still Giffhorn's algorithm was a big step, as it allowed – for the first time – low nondeterminism, while basically maintaining the LSOD approach.

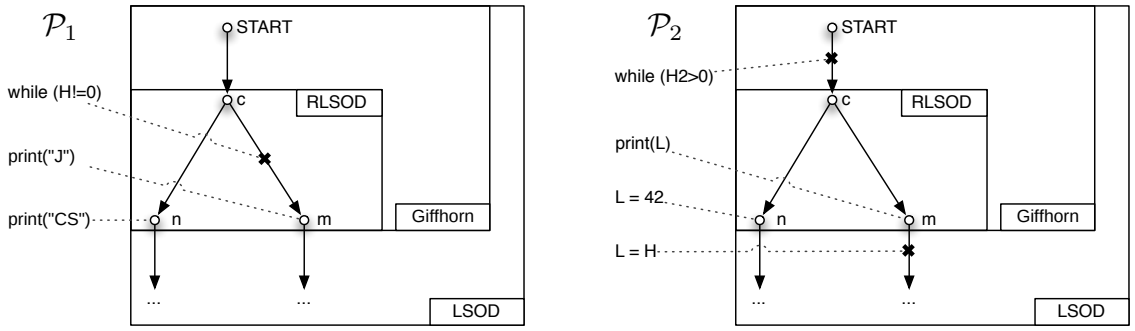


Fig. 6. Visualization of LSOD vs. Giffhorn's criterion vs. RLSOD (condition 3 / 3' / 3"). CFGs for Figure 1 right resp. Figure 4 right are sketched. *n*/*m* produces low nondeterminism, *c* is the common dominator. LSOD prohibits any low nondeterminism; Giffhorn allows low nondeterminism which is not reachable by any high events; RLSOD allows low nondeterminism which may be reached by high events if they are before the common dominator. The marked regions are those affected by low nondeterminism; inside these regions no high events are allowed. Thus RLSOD is much more precise.

5. RLSOD

In the following, we will generalize conditions 2' and 3' to obtain the much more precise RLSOD criterion.

To motivate the improvement, consider again Figure 1 right (program $\mathcal{P}_1$) and Figure 4 right (program $\mathcal{P}_2$). Both contain race conditions influencing low behavior, $\mathcal{P}_1$ through low-nondeterminism and $\mathcal{P}_2$ through a data race. When comparing $\mathcal{P}_1$ and $\mathcal{P}_2$, a crucial difference comes to mind. In $\mathcal{P}_2$ the troublesome high statement can reach *both* statements forming the race condition, whereas in $\mathcal{P}_1$, the high statement can reach only one of them. In both programs some loop running time depends on a high value, but in $\mathcal{P}_2$, the subsequent statements are influenced by this "timing leak" in exactly the same way, while in $\mathcal{P}_1$ they are not.

In terms of the PN definition, remember that $\mathcal{P}_1$ has only two low classes
$[t]^1_L = \{t' \mid t' = \ldots \texttt{print("J")} \ldots \texttt{print("CS")} \ldots\}$ and
$[t]^2_L = \{t' \mid t' = \ldots \texttt{print("CS")} \ldots \texttt{print("J")} \ldots\}$. Likewise, $\mathcal{P}_2$ has two low classes
$[t]^1_L = \{t' \mid t' = \ldots \texttt{print(42)} \ldots\}$ and
$[t]^2_L = \{t' \mid t' = \ldots \texttt{print(0)} \ldots\}$, depending on the outcome of the data race. The crucial difference is that for $\mathcal{P}_1$, the probability for the two classes under i resp. i' is not the same (see above), but for $\mathcal{P}_2$, $P_i([t]^{1,2}_L) = P_{i'}([t]^{1,2}_L)$ holds!

Technically, $\mathcal{P}_2$ contains a point c which *dominates* both racing statements $n \equiv \texttt{L} = \texttt{42};$ and $m \equiv \texttt{print(L)}$, and all relevant high events always happen before c. Domination means that any control flow from *START* to n or m must pass through c. In $\mathcal{P}_2$, c is the point immediately before the first fork. In contrast, $\mathcal{P}_1$ has only a trivial common dominator for the low nondeterminism, namely the *START* node, and on the path from *START* to $n \equiv \texttt{print("J")}$ there is no high event, while on the path to $m \equiv \texttt{print("CS")}$ there is.

Intuitively, the high inputs can cause strong nondeterministic high behaviour, including stuttering. But if LSOD conditions 1 + 2 are always satisfied, and if there are no high events in any trace between c and n resp. m, the effect of the high behaviour is always the same for n and m and thus "factored out". It cannot cause a probabilistic leak – the dominator "shields" the low nondeterminism from high influence. Note that $\mathcal{P}_2$ contains an additional high statement $m' \equiv \texttt{read(H)}$ but that is behind n (no control flow is possible from m' to n) and thus cannot influence the n/m nondeterminism.

5.1. Improving Conditions 2' and 3'

The above example has demonstrated that low nondeterminism may be reachable by high events without harm, as long as these high events always happen before the common dominator of the nondeterministic low statements. This observation will be even more important if dynamically created threads are allowed (as in JOANA, cmp. Section 7). We will now provide precise definitions for this idea.

Definition 13 (Common dynamic ancestor)**.** *Let $n, m, c \in N$ be statements.*

(1) c is a dominator for n, written c dom n, if c occurs on every CFG path from START to n.
(2) c is a common dominator for n, m, written c cdom (n, m), if c dom $n \wedge$ c dom m.
(3) c is a common dynamic ancestor for n, m, written c cda (n, m), if
c cdom $(n, m) \wedge \neg MHP(c, n) \wedge \neg MHP(c, m)$.
(4) If c cda (n, m) and $\forall c'$ cda (n, m): c' dom c, then c is called an immediate common dynamic ancestor.

Efficient algorithms for computing dominators can be found in many compiler textbooks. They can be extended to calculate common dynamic ancestors. The stricter definition of common dynamic ancestors compared to common dominators is needed to deal with the possibility that the same thread can be spawned several times, as e.g. in our case study (chapter 7). Note that *START* itself is a (trivial) common dynamic ancestor for every n, m. RLSOD works with any common dynamic ancestor. We thus assume a function *cda* which for every statement pair returns a common dynamic ancestor, and write $c = cda(n, m)$. Note that the implementation of *cda* may depend on the precision requirements, but once a specific *cda* is fixed, c depends solely on n and m. Straightforward implementation of the RLSOD idea leads to the following rules, replacing rules 3' and 2' from Giffhorn's optimized criterion:

3". $\forall n, n' : \ lnd(n, n') \wedge c = cda(n, n') \implies \forall n'' \in (path(c, n) \cup path(c, n')) \cap I : ucl(n'') = L$
2". $\forall n, n' \in N, n'' \in O : race(n, n') \wedge c = cda(n, n') \wedge ucl(n'') = L \wedge n' \in BS(n'')$
$\implies \forall n''' \in (path(c, n) \cup path(c, n')) \cap I : ucl(n''') = L$

These conditions are most precise (generate the least false alarms) if *cda* returns the immediate common dynamic ancestor, because in this case it demands that $cl(n'') = L$ for the smallest set of nodes "behind" the common ancestor.[8] Figure 6 illustrates the RLSOD definition (condition 3"). Note that Giffhorn's optimized criterion trivially fulfils conditions 2" and 3", where *cda* always returns *START*. In [2], we provided a soundness proof for the case of just one low-nondeterminism. However, as the next section will show, in the case of multiple low-nondeterministic statements those rules are not sound.

5.2. Classification Revisited

Consider the program in Figure 7 middle/right. This example contains a probabilistic leak as follows. `H` influences the running time of the first while loop, hence `H` influences whether line 10 or line 18 is performed first. The value of `tmp2` influences the running time of the second loop, hence it also influences whether `L1` or `L2` is printed first. Thus `H` indirectly influences the execution order of the final print statements. Indeed the program does not fulfill Giffhorn's criterion, as the print statements can be

[8] Note that in programs with procedures and threads, immediate dynamic ancestors may not be unique due to context-sensitivity [34].

```
1  void thread1():
2    tmp = 1;
3    if (H) {
4       tmp = 100;
5    }
6    fork thread2();
7    while (tmp > 0) {
8       tmp = tmp − 1;
9    }
10   tmp2 = 1;
11   fork thread3();
12   while (tmp2 > 0) {
13      tmp2 = tmp2 − 1;
14   }
15   print(L1);
16 void thread2():
17   tmp2 = 100;
18
19 void thread3():
20   print(L2);
```

Fig. 7. A leak which goes undiscovered if classification of statements is incomplete.

reached from the high statement in line 3 (middle). Applying 3", the common dynamic ancestor for the two print statements is line 10. But the only input statement is line 3, which is before the cda.

The classification of line 10 is thus crucial. If we had $cl(10) = H$, then this classification propagates in the PDG (due to the flow equation $cl(n) = \bigsqcup_{m \to n} cl(m)$) and lines 12/13 are classified high. RLSOD is violated, and the probabilistic leak discovered.

Hence we use the flow equation to calculate the classification propagation from line 3 to line 10, and then 12/13. Only line 3 is explicitly high and only lines 4, 7, 8 are PDG-reachable from 3. Thus $cl(10) = L$. Hence RLSOD would be satisfied because 3,4,7,8 are *before* the common dominator. The leak would go undiscovered! The problem is that rules 2" and 3" are not applied recursively. To capture the effect of recursive influence through nondeterminism, the flow equations must be extended.

In general, the rule is as follows. The standard flow equation $cl(n) = \bigsqcup_{m \to n} cl(m)$ expresses the fact that if a high value can reach a PDG node m upon which n is dependent, then the high value can also reach n. Likewise, if there is low nondeterminism with $\mathrm{MHP}(n,m)$, and $cda(n,m) = c$, and the path $c \to^*_{CFG} n$ violates RLSOD – that is, it contains high statements – then the high value can reach n. Thus $cl(n) = H$ must be enforced. This rule must be applied recursively until a fixpoint is reached.

Definition 14 (Classification in PDGs)**.** *A PDG $G = (N, \to)$ is classified correctly, if*

(a) $\forall n \in N\colon cl(n) \geqslant \bigsqcup_{m \to n} cl(m)$,
(b) $\forall n, m \in N\colon MHP(n,m) \wedge c = cda(n,m) \wedge \exists c' \in path(c,n), cl(c') = H \implies cl(n) = H$.
(c) $\forall n \in I : ucl(n) = cl(n)$ *and* $\forall n \in O : ucl(n) \geqslant cl(n)$.

In condition (a), $\geqslant$ must be used because (b) can push $cl(n)$ higher than $\bigsqcup_{m \to n} cl(m)$. Condition (b) can be rewritten as $MHP(n,m) \wedge c = cda(n,m) \implies cl(n) \geqslant \bigsqcup_{c' \in path(c,n)} cl(c')$; making it formally similar to (a).[9] Note the asymmetry in condition (c): Treating a public input value as secret is sound, thus one might want to use $\forall n \in I : ucl(n) \leqslant cl(n)$ in that condition. However, we assume that the attacker can observe when low input statements are executed. Thus, whether they are executed must not depend on H values, so $\forall n \in I : ucl(n) \geqslant cl(n)$ must be enforced as well.

[9]This formulation is used by JOANA; it also has the advantage that RLSOD can be used for an arbitrary security lattice. Note that our current soundness proof works only with the L/H lattice.

For a sequential program, *MHP* is always false, so only rules (a) and (c) remain. But (a)+(c) is equivalent to definition 9, the calculation of sequential noninterference. In Figure 7 middle/right, definition 14 enforces line 10 to be classified high, as we have $cda(10, 18) = 6$, and on the path from 6 to 10, lines 7 and 8 are high.

Now, classification rules (a), (b) and (c) could be combined with 1", 2" and 3" to yield a sound RLSOD criterion. But surprisingly it turns out that this is not necessary – section 6 will demonstrate that classifiability according to definition 14 already implies PN! In fact, 1", 2" and 3" are necessary conditions for classifiability:

Theorem 4. *If the program can be classified according to definition 14, then the RLSOD conditions 1", 2", 3" hold.*

Proof. Assume that (a), (b), (c) hold, but one of the rules 1", 2", 3" is violated. We therefore have 3 cases:

(1) 1" is violated: Since an H source n is in the backward slice of an L sink n', with classification rule (c) we have $cl(n) = H$, $cl(n') = L$ and $n \to^* n'$. But then repeated application of rule (a) demands $cl(n') = H$, which is a contradiction.
(2) 3" is violated: For the two *lnd* statements n and n' we have $cl(n) = cl(n') = L$ by rule (c). But then we have a statement n'' with $cl(n'') = H$ on the control flow path from $c := cda(n, n')$ to n or to n'. Without loss of generality we assume that n'' lies on the path from c to n. But then rule (b) demands $cl(n) = H$, contradicting $cl(n) = L$.
(3) 2" is violated: We have $cl(n'') = L$ by rule (c), and by rule (a) we have $cl(n') = L$ since $n' \in BS(n'')$. Since we also have $race(n, n')$, n must write the same variable that n' reads or writes, and thus we have $n \in BS(n')$ and $cl(n) = L$ as well. Then we can apply the same argument to n and n' as in the previous case.

□

Henceforth, we will subsume definition 14 under the notion of RLSOD. To actually use it as a PN checker, it uses a fixpoint iteration similar to the sequential one (definition 9, see also [12, 33]).

6. The General Soundness Proof

In the conference paper preceding this article [2], a soundness proof was provided for the special case that there is just one occurrence of low-nondeterminism (i.e. $|\{(n, n') \mid lnd(n, n') \vee race(n, n')\}| \leqslant 1$). The full proof posed more difficulties than expected, but eventually led to a simpler formulation of the RLSOD criterion, namely in form of Definition 14. We thus omit the proof of the special case (see [2], sec. 4.3), but will in this section describe the full soundness proof, based on Definition 14. As in [2], the proof relies on the notion of conditional probability for traces.

Definition 15. *Let $t_1 \cdots$ be the set of traces beginning with prefix t_1, so that $P_i(t_1 \cdots) = \sum_{t=t_1 \cdot t_2} P_i(t)$ is the probability that execution under input i begins with t_1. For a set T of traces let $T \cdots = \bigcup_{t \in T} t \cdots$. We denote with $P_i(t_2 \mid t_1)$ the conditional probability that after t_1, execution continues with t_2; we have*

$$P_i(t_2 \mid t_1) = P_i(t_1 \cdot t_2) / P_i(t_1 \cdots)$$

This notion extends to sets of traces:

$$P_i(T' \mid T) = P_i(T \cdot T')/P_i(T\cdots) = \sum_{t\in T\cdot T'} P_i(t) / \sum_{t\in T\cdots} P_i(t)$$

In the following it will always hold that $T(i) \cap T\cdots \neq \emptyset$, hence $\sum_{t\in T\cdots} P_i(t) \neq 0$.

The following soundness theorem and its derivation will of course be based on Definition 3 (low- equivalent traces), but in the proofs we will also need a variant of this definition as an auxiliary construction: We define $E_{cl}((s,o,s'))$, $t \sim_{cl} t'$ and $[t]_{cl}$ exactly as $E_L((s,o,s'))$, $t \sim_L t'$ and $[t]_L$ (see Definition 3), except that in the definition *ucl* is replaced by *cl*. We omit a repetition of the definition details, as E_{cl}, $\sim_{cl}$ and $[]_{cl}$ are only used in the proofs and do not appear in the soundness theorem itself.

Lemma 3. *(1)* $[t_1]_{cl}[t_2]_{cl} = [t_1t_2]_{cl}$
(2) $P_i([t]_{cl}) = P_i([\varepsilon]_{cl} \mid [t]_{cl}) \cdot P_i([t]_{cl}\cdots \mid [\varepsilon]_{cl})$
(3) $P_i([t_1t_2]_{cl}\cdots \mid [\varepsilon]_{cl}) = P_i([t_2]_{cl}\cdots \mid [t_1]_{cl}) \cdot P_i([t_1]_{cl}\cdots \mid [\varepsilon]_{cl})$

Proof. (1) We have $LS_{cl}(t\cdot t') = LS_{cl}(t)\cdot LS_{cl}(t')$ from Definition 3 for all t,t' since LS_{cl} is a composition of *map* and $filter$. With the definition of $\sim_{cl}$, this implies $[t_1]_{cl}[t_2]_{cl} = [t_1t_2]_{cl}$.

(2) $P_i([\varepsilon]_{cl} \mid [t]_{cl}) \cdot P_i([t]_{cl}\cdots \mid [\varepsilon]_{cl}) = P_i([\varepsilon]_{cl}[t]_{cl})/P_i([t]_{cl}\cdots)\cdot P_i([t]_{cl}\cdots)/P_i([\varepsilon]_{cl}\cdots)$
$= P_i([\varepsilon]_{cl}[t]_{cl})/P_i([\varepsilon]_{cl}\cdots) = P_i([t]_{cl})$,
using equation 1 and the fact that all traces begin with $\varepsilon \in [\varepsilon]_{cl}$, so $P_i([\varepsilon]_{cl}\cdots) = 1$.

(3) $P_i([t_2\cdots]_{cl} \mid [t_1]_{cl}) \cdot P_i([t_1\cdots]_{cl} \mid [\varepsilon]_{cl}) = P_i([t_1]_{cl}[t_2]_{cl}\cdots)/P_i([t_1]_{cl}\cdots)\cdot P_i([t_1]_{cl}\cdots)/P_i([\varepsilon]_{cl})$
$= P_i([t_1]_{cl}[t_2]_{cl}\cdots)/P_i([\varepsilon]_{cl}) = P_i([t_1]_{cl}[t_2]_{cl}\cdots \mid [\varepsilon]_{cl}) = P_i([t_1t_2]_{cl}\cdots \mid [\varepsilon]_{cl})$
□

For the following theorems, let $op(c)$ denote the operation of an event c, i.e. for $c = (s,o,s')$, we have $op(c) = o$. For convenience, we first prove the following lemmas:

Lemma 4. *Let $\mathcal{P}$ be a terminating program. Assume that classification rules (a) and (c) according to Definition 14 hold. Let $t_1\cdot c\cdot t_2 \in T(i)$ and $t_1'\cdot c'\cdot t_2' \in T(i')$ with $op(c) = op(c')$, $cl(op(c)) = L$, $t_1 \sim_{cl} t_1'$ and $i \sim_L i'$.*
Then $E_{cl}(c) = E_{cl}(c')$.

Proof. Let $o := op(c) = op(c')$. If o reads input, we have $stmt(o) \in I$, and $ucl(stmt(o)) = cl(stmt(o)) = L$ by rule (c), and it therefore reads from the low input stream. From rule (c) we also have that all other operations that have read from this stream before are classified as low, and thus show up in *cl*-low-observable subtraces. This means the same reads must have happened before c and c', and so c and c' read the same values from input.

If o reads values from memory, o is data dependent on the operations that wrote those values, so those must be classified as L by classification rule (a). But then those show up in a *cl*-low-observable subtrace, and since $t_1 \sim_{cl} t_1'$, the same of those operations must have happened for t_1 and t_1', and they wrote the same values in the same order. Therefore, c and c' read the same values. Since statements themselves are deterministic, the values written by c and c' are equal as well. Thus we have $E_{cl}(c) = E_{cl}(c')$. □

Lemma 5. *Let $\mathcal{P}$ be a program that always terminates and let classification rule (a) according to Definition 14 hold. Let $t = t_1 \cdot c \cdot t_2$ be a possible trace for $\mathcal{P}$ and $cl(c) = L$. Let $t' = t'_1 \cdot t'_2$ with $t'_1 \sim_{cl} t_1$ also be a possible trace.*

Then $op(c)$ must occur on t'_2.

Proof. Since operations are unique in a trace, t_1 cannot contain $op(c)$, and neither can t'_1 since $cl(c) = L$. Thus, it suffices to show that $op(c)$ is executed on $t'_1 \cdot t'_2$. This is trivially fulfilled for the starting operation. Else, let p be the operation that directly controls whether $op(c)$ is executed. We have $stmt(c) = stmt(p)$ or $stmt(c)$ is control dependent on $stmt(p)$. With classification rule (a), we have $cl(p) = L$. Since $t'_1 \sim_{cl} t_1$, p was executed on t'_1 as well and has read the same values as on t_1. Therefore, it chooses the same branch, so $op(c)$ must occur on $t'_1 \cdot t'_2$ as well. □

We now show that if a program's PDG can be classified correctly, then for a *cl*-low-class, the probability of observing it does not differ between low-equivalent inputs:

Theorem 5. *Let $\mathcal{P}$ be a program that always terminates and ucl a user annotation for $\mathcal{P}$. Let cl be a correct classification of its PDG according to Definition 14.*

Now let i, i' be two inputs with $i \sim_L i'$, let t be a trace. Then

$$P_i([t]_{cl}) = P_{i'}([t]_{cl}).$$

Proof. We will prove the equation by contradiction. So, let us assume $P_i([t]_{cl}) \neq P_{i'}([t]_{cl})$.

Let $t = c_1 \cdot c_2 \cdots c_n$ be the decomposition of t into single events. With the calculation rules from Lemma 3, applying rule 2 and then repeatedly applying rule 3, we get

$$P_i([t]_{cl}) = (\prod_{j=1}^{n} P_i([c_j]_{cl} \cdots \mid [c_1 \cdot \ldots \cdot c_{j-1}]_{cl})) \cdot P_i([\varepsilon]_{cl} \mid [t]_{cl}),$$

and the same for i'. Since the left sides for i and i' are not equal, the same must hold for at least one factor. Furthermore, we have

$$P_i([\varepsilon]_{cl} \mid [t]_{cl}) = P_{i'}([\varepsilon]_{cl} \mid [t]_{cl})$$

by the following argument: If not both sides are 1, there is a low event that is possible after a trace in $[t]_{cl}$. From Lemma 5 we then have that this event occurs on every trace $t' \sim_{cl} t$, making both probabilities 0.

Thus, there is a j that the factor

$$P([c_j]_{cl} \cdots \mid [c_1 \cdots c_{j-1}]_{cl})$$

is different for i and i'. Let $t_1 = c_1 \cdots c_{j-1}$ and $c = c_j$. Then we have

$$P_i([c]_{cl} \cdots \mid [t_1]_{cl}) \neq P_{i'}([c]_{cl} \cdots \mid [t_1]_{cl}).$$

In the following, we will focus on these two conditional probabilities, so we can assume that the trace that has happened until that point is *cl*-low-equivalent to t_1.

Without loss of generality we assume

$$P_i([c]_{cl} \cdots \mid [t_1]_{cl}) > P_{i'}([c]_{cl} \cdots \mid [t_1]_{cl})$$

(switch i and i' if necessary). $op(c)$ gets executed for i since the former probability is greater than zero, and we have $cl(c) = L$ because else both probabilities would be equal to 1. With Lemma 5 we have that $op(c)$ will be executed in every trace in $[t_1]_{cl} \cdots$.

Since the remaining trace must execute $op(c)$ given an execution in $[t_1]_{cl}$ has happened, its cl-low-observable part cannot be empty. The conditional probabilities for the different cl-low-observable parts of low events given $[t_1]_{cl}$ must add up to 1 for i and i', but we have

$$P_i([c]_{cl} \cdots \mid [t_1]_{cl}) > P_{i'}([c]_{cl} \cdots \mid [t_1]_{cl}).$$

Thus, there is a c' with $E_{cl}(c) \neq E_{cl}(c')$ such that

$$P_i([c']_{cl} \cdots \mid [t_1]_{cl}) < P_{i'}([c']_{cl} \cdots \mid [t_1]_{cl}).$$

Analogously to above, $op(c')$ must happen after each trace in $[t_1]_{cl}$.

If $op(c) = op(c')$, with Lemma 4 we get $E_{cl}(c) = E_{cl}(c')$, contradicting $E_{cl}(c) \neq E_{cl}(c')$. Thus $op(c) \neq op(c')$. Let $o := op(c)$ and $o' := op(c')$. o and o' must be executed after a trace in $[t_1]_{cl}$ has been executed. In fact, o can be executed directly after it (at least for input i) and since $o \neq o'$, o' must then happen after o. Analogously, o can happen after o'. This makes $s := stmt(o)$ and $s' := stmt(o')$ an MHP-pair. Let $d := cda(s, s')$. Then by classification rule (b) with $n = s, m = s'$ all nodes in $path(d, s)$ are classified as L. o can be scheduled for input i, so the CFG predecessors of s that were already executed allow o to be executed next. Those are in $path(d, s)$ since otherwise we would have $d = s$, which is impossible because with $\mathrm{MHP}(s, s')$ we have $\mathrm{MHP}(d, s')$, a contradiction to Definition 13. Thus, they are cl-low-observable and therefore have happened for any trace $t_1' \sim_{cl} t_1$ as well. Thus, o can be scheduled for every such trace t_1', regardless of i or i'. The same argument shows that o' can be scheduled after every such t_1'. From Lemma 4 we also have that the only possible cl-low-observable part of the event for o is $E_{cl}(c)$, so

$$P_i([c]_{cl} \cdots | [t_1]_{cl}) = P_i([o]_{cl} \cdots | [t_1]_{cl})$$

and

$$P_i'([c]_{cl} \cdots | [t_1]_{cl}) = P_i'([o]_{cl} \cdots | [t_1]_{cl})$$

(we use $[o]_{cl}$ here as a shorthand for $\bigcup_{op(c)=o} [c]_{cl}$). The same holds for o' and c'. If we go from i to i', the probability of scheduling o gets smaller but the probability of scheduling o' gets greater. Therefore, the relative scheduling probabilities do not stay the same, even though for both inputs, both operations can be scheduled. This contradicts the assumption of a truly probabilistic scheduler. □

We now can prove that the conditions of the previous theorem guarantee PN.

Corollary 1. *Let $\mathcal{P}$ be a terminating program and ucl a user annotation for $\mathcal{P}$. Let cl be a correct classification of its PDG according to Definition 14.*

Now let $i \sim_L i'$ according to Definition 5, let $t \in \Theta$. Then

$$P_i([t]_L) = P_{i'}([t]_L).$$

Proof. Classification rule (c) ensures that all sources and sinks n with $ucl(n) = L$ also fulfill $cl(n) = L$. Thus, for all traces t_1, t_2 we have $t_1 \sim_{cl} t_2 \implies t_1 \sim_L t_2$. Therefore, we can write $[t]_L$ as disjoint union of all $[t']_{cl} \subseteq [t]_L$. With the probability formula for disjoint unions we have

$$P_i([t]_L) = \sum_{[t']_{cl} \subseteq [t]_L} P_i([t']_{cl}),$$

and the same for i'. By applying Theorem 5, we get $P_i([t']_{cl}) = P_{i'}([t']_{cl})$ for all $[t']_{cl}$. Thus,

$$P_i([t]_L) = \sum_{[t']_{cl} \subseteq [t]_L} P_i([t']_{cl}) = \sum_{[t']_{cl} \subseteq [t]_L} P_{i'}([t']_{cl}) = P_{i'}([t]_L)$$

□

7. Case Study: E-Voting

We will now apply RLSOD to an experimental e-voting system developed in collaboration with R. Küsters et al. This system aims at a provably secure e-voting software that uses cryptography to ensure *computational indistinguishability*. To prove computational indistinguishability, the cryptographic functions are replaced with an "ideal variant": The encryption creates a random number as encrypted message and remembers the connection of this message and the key to the plaintext. The decryption can then get the plaintext back by providing the key and the encrypted text.[10] It is then checked by IFC that no flow exists between plain text, secret key and encrypted message; that is, probabilistic noninterference holds for the e-voting system with ideal crypto implementation. By a theorem of Küsters, noninterference of the ideal variant implies computational indistinguishability for the system with real encryption [14, 15].

The example uses a multithreaded client-server architecture to send encrypted messages over the network. It consists of 550 LoC with 16 classes. The interprocedural control flow is sketched in Figure 8; Figure 9 contains relevant parts of the code. The main thread starts in class `Setup` in line 2ff: First it initializes encryption by generating a private and public key, then it spawns a single `Server` thread before entering of the main loop. Inside the main loop it reads a secret message from the input and spawns a `Client` that takes care of the secure message transfer: The client encrypts the given message and subsequently sends it via the network to the server. Note that there are multiple instances of the client thread as a new one is started in each iteration.

There are two sources of secret (*high*) information: (1) the value of the parameter `secret_bit` (line 2) that decides about the content of the message; and (2) the private key of the encryption (line 30).

[10] Note that due to this construction, at the encryption site there is no flow from the plaintext or key to the encrypted message.

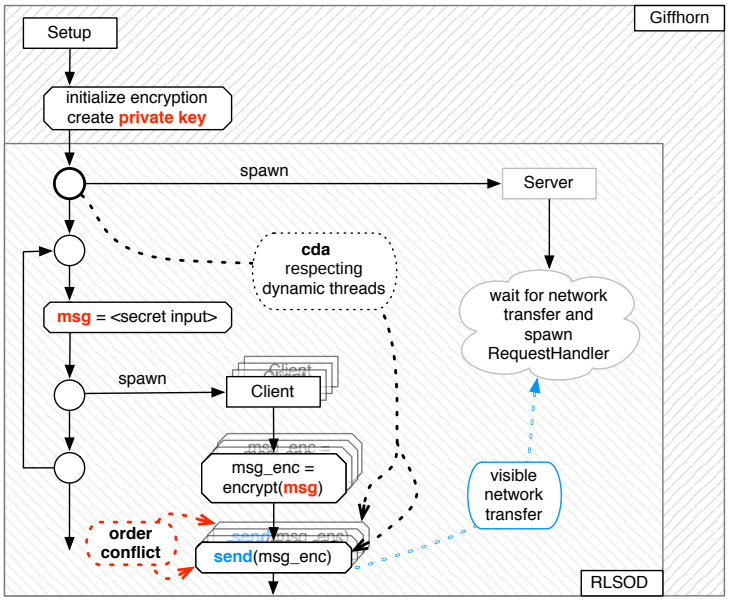


Fig. 8. CFG structure of the multithreaded server-client based message transfer.

Both are marked for JOANA with a `@Source` annotation, signalling an input statement (with a default security level of *H*). By Definition 14, (2) propagates to lines 39, 41, 5, and 9 which are also classified High. Likewise, (1) propagates to lines 19 and 22, which are thus High as well.

As information sent over network is visible to the attacker, calls to the method `sendMessage` (line 57f) are marked as a *low* `@Sink`, signalling an output statement (with a default security level of *L*). JOANA was started in "Giffhorn" mode, and – analysing the "ideal variant" – immediately guarantees that there are no explicit or implicit leaks. However the example contains two potential probabilistic leaks, which are both discovered by JOANA using Giffhorn's criterion; one is later uncovered by RLSOD to be a false alarm.

To understand the first leak in detail, remember that this e-voting code spawns new threads in a loop. This will cause low-nondeterminism because the running times for the individual threads may vary and thus their relative execution order depends on scheduling. This low-nondeterminism is (context-sensitively) reachable from the high private-key initialization in line 39, hence criterion 2'/3' will cause an alarm. Technically, we have $\mathrm{MHP}(57,57) \wedge \mathit{ucl}(57) = L$; that is, line 57 is low-nondeterministic with itself (because the same thread is spawned several times). Furthermore, $START \rightarrow^*_{CFG} 39 \rightarrow^*_{CFG} 57 \wedge \mathit{ucl}(39) = H$. Thus criterion 3' is violated: Giffhorn's criterion (as well as classical LSOD) thinks there is a probabilistic leak.

Now let us apply RLSOD to this leak. The common dynamic ancestor for all the low-nondeterministic message sends in line 57 is located just before the loop header: $11 = \mathit{cda}(57,57)$.[11] Now it turns out that the initialisation of private keys lies *before* this common dynamic ancestor: lines 30, 39, 41, 5, 8, and 9 context-sensitively dominate line 11, and cannot happen parallel to it. Thus by RLSOD criterion 3", this potential leak is uncovered to be a false alarm: the private key initialisation is in fact secure!

[11] Note that we indeed need *cda* instead of *cdom* here, such that the static *cda* lies before *all* dynamically possible spawns. JOANA handles such situations correctly, as well as handling interprocedural, context-sensitive dynamic ancestors.

```
1  public class Setup {
2    public static void setup(@Source boolean secret_bit) {
3      // HIGH input
4      // Public-key encryption for Server
5      Decryptor serverDec = new Decryptor();
6      Encryptor serverEnc = serverDec.getEncryptor();
7      // Creating the server
8      Server server = new Server(serverDec, PORT);
9      new Thread(server).start();
10     // The adversary decides how many clients we create
11     while (Environment.untrustedInput() != 0) {
12       // determine the value the client encrypts:
13       // the adversary gives two values
14       byte[] msg1 = Environment.untrustedInputMessage();
15       byte[] msg2 = Environment.untrustedInputMessage();
16       if (msg1.length != msg2.length) { break; }
17       byte[] msg = new byte[msg1.length];
18       for(int i = 0; i < msg1.length; ++i)
19         msg[i] = (secret_bit ? msg1[i] : msg2[i]);
20       // spawn new client thread
21       Client client = new Client(serverEnc, msg, HOST, PORT);
22       new Thread(client).start();
23     }
24   }
25 }
26
27 public class KeyPair {
28   public byte[] publicKey;
29   @Source
30   public byte[] privateKey; // HIGH value
31 }
32
33 public final class Decryptor {
34   private byte[] privKey;
35   private byte[] publKey;
36   private MessagePairList log = new MessagePairList();
37   public Decryptor() {
38     // initialize public and secret (HIGH) keys
39     KeyPair keypair = CryptoLib.pke_generateKeyPair();
40     publKey = copyOf(keypair.publicKey);
41     privKey = copyOf(keypair.privateKey);
42   }
43   ...
44 }
45
46 public class Client implements Runnable {
47   private byte[] msg;       private Encryptor enc;
48   private String hostname;  private int port;
49   ...
50   @Override
51   public void run() {
52     // encrypt
53     byte[] msg_enc = enc.encrypt(msg);
54
55     // send
56     long socketID = Network.openConnection(hostname, port);
57     Network.sendMessage(socketID, msg_enc);
58     Network.closeConnection(socketID);
59   }
60 }
61
62 public class Network {
63   @Sink // LOW output
64   public static void sendMessage(long socketID, byte[] msg) throws NetworkError {
65     ...
66   }
67   ...
68 }
```

Fig. 9. Relevant parts of the multithreaded encrypted message passing system with security annotations for JOANA.

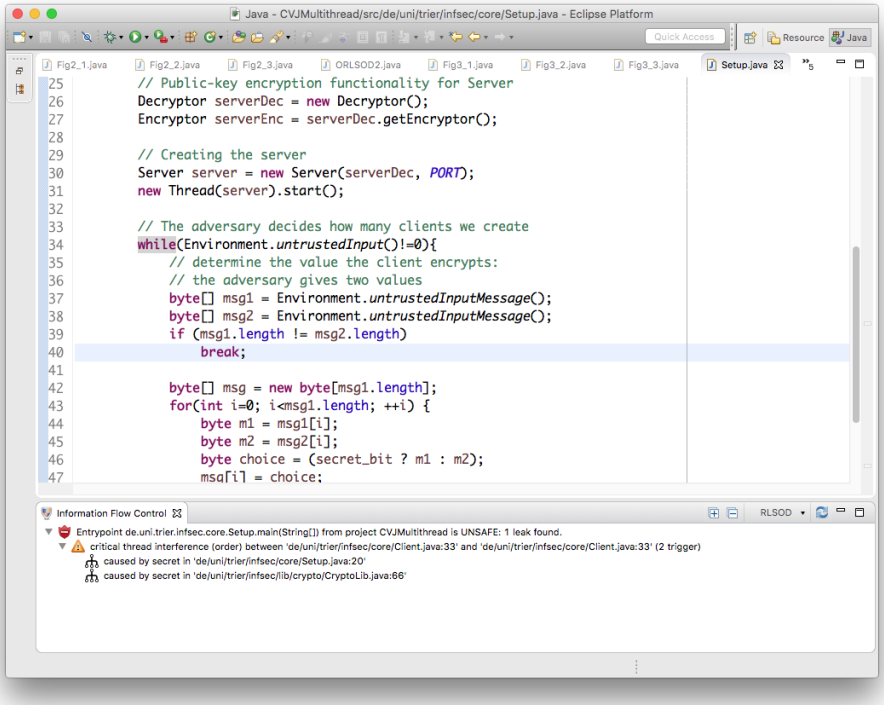


Fig. 10. JOANA analysing the e-voting source code

The second potential probabilistic leak comes from the potential high influence by `secret_bit` in line 19 to the low-nondeterministic message sends in line 63. Technically, we have the PDG High chain $2 \to 19 \to 21 \to 53 \to 57$, but 57 is manually classified Low. However this second leak candidate is not eliminated by RLSOD, and indeed is a probabilistic leak: since the `encrypt` run time may depend on the message, the scheduler will statistically generate a specific "average" order of message send executions (remember the scheduler must be probabilistic). An attacker can thus watch this execution order, and deduce information about the secret messages. Technically, this subtle leak is discovered by RLSOD because the high operation which accesses the secret bit lies *behind* the common dynamic ancestor, but before the low-nondeterminism:

$$11 = cda(57, 57) \to^*_{CFG} 19 \to^*_{CFG} 57.$$

JOANA must and will report this probabilistic leak. The engineer might however decide that the leak is not dangerous. If the engineer can guarantee that the `encrypt` run time does *not* depend on `msg`, the leak may be ignored.

JOANA detects both potential probabilistic leaks in about 5 seconds on a standard PC (including PDG construction). A JOANA screenshot showing the analysis of the e-voting source code is given in Figure 10; more details on the JOANA GUI can be found in [17]. After JOANA is set to RLSOD, one of the leaks disappears as described above. We consider the e-voting example a rather spectacular example how RLSOD improves precision. Note however that a systematic evaluation of RLSOD precision has not yet been tackled, as it is difficult to find realistic example programs with probabilistic leaks.

	#BC Instr.	PDG time	PDG time O-S
wallet	116	504	924
clientserver	601	204	298
battleship	937	408	1,078
corporatecard	1,186	427	821
safeappplet	1,295	376	1,332
hybrid	1,479	454	729
cloudstorage	1,972	563	1,030
j2mesafe	5,519	3,937	–
purse	10,807	6,004	13,165
barcode	11,406	1,307	–
onetimepass	13,034	48,729	–
bexplore	14,041	21,538	–
keepass	16,427	9,495	–
freecs	49,396	1,301,072	–
hsqldb	126,860	4,129,274	–

Table 1
PDG construction times (in milliseconds) for benchmark programs

8. Evaluation

Table 1 presents PDG construction times for various programs.[12] These times are taken from a recent JOANA scalability study provided by J. Graf in [33]. All measurements took place on a machine with a Core i7 processor, 32GB RAM, with Java 8 64-Bit. For the 15 benchmark programs, program size in byte code instructions is given (without library functions), as well as PDG construction time in milliseconds. "PDG time O-S" is PDG construction time with object-sensitive points-to analysis; this variant is more expensive but much more precise. The largest program is "hsqldb" with 126860 byte code instructions, which corresponds to about 65kLOC source code.

[33] provides additional data on PDG size, summary edges, impact of points-to analysis details, and more. RLSOD time is not included in the PDG construction times. One observes that runtimes are strongly nonlinear, but moderate (1-2 hours/50kLOC). Memory is the limiting factor (7 of the programs cannot be analysed using object-sensitive points-to with 32GB). Note that JOANA offers many analysis options, in particular choices for MHP and points-to precision [17].

Table 2 shows the runtimes of LSOD, Giffhorn's criterion and RLSOD for a subset of the programs in table 1, "BarrierBench" from the Java Grande Benchmark and the e-voting program from section 7; on the same machine configuration as above. PDG construction time is not included. PDGs were built with object graphs and instance-based points-to precision (cmp. [33]). For each $n = 1, 5, 10$ we selected n sources and n sinks randomly from the PDG nodes. For each program, criterion and n we performed 10 measurements. The table shows the average runtime in milliseconds. For the top four programs, we used a more precise and more expensive MHP analysis. Since it does not scale for the last two programs, we used a simple MHP analysis there. Note that this choice only affects precision, not soundness. For the small programs, all (R)LSOD times are less than 100 msec; for the largest program (R)LSOD times are below 2 min. Interestingly, for the largest program and $n > 1$, "Giffhorn" is a little faster than LSOD, and RLSOD is 4 times faster. The reason is that both the LSOD and Giffhorn implementations use multiple

[12] Links to all benchmark programs can be found at `https://pp.ipd.kit.edu/projects/joana/rlsod-jcs.php`.

Program	#BC Instr.	LSOD			Giffhorn			RLSOD		
		1	5	10	1	5	10	1	5	10
barrierbench	416	38	31	32	38	31	34	93	63	55
battleship	937	11	17	24	13	17	21	25	17	14
evoting	1,106	13	9	11	13	10	12	37	23	21
barcode	11,406	20	21	25	25	23	28	58	37	36
freecs	49,396	23,945	34,502	50,427	23,622	32,072	45,160	54,280	53,042	56,111
hsqldb	126,860	16,427	70,740	133,777	17,288	62,581	119,767	31,885	30,333	30,333

Table 2
Performance of LSOD, Giffhorn's criterion and RLSOD with different numbers of sources/sinks (in milliseconds)

backward slices according to definition 12 resp. conditions 2' / 3', while RLSOD is based on definition 14 and needs no backward slices. This also explains why RLSOD does not really depend on n.

9. Related Work

Zdancewic's work [8] was the starting point for us, once Giffhorn discovered that the Zdancewic LSOD criteria can naturally be checked using PDGs. Zdancewic uses an interesting definition of low-equivalent traces: low equivalence is not demanded for traces, but only for every subtrace for every low variable ("location traces"). This renders more traces low-equivalent and thus increases precision. But location traces act contrary to flow-sensitivity (relative order of variable accesses is lost), and according to our experience flow-sensitivity is essential.

While strict LSOD guarantees probabilistic non-interference for any scheduler, it is too strict for multi-threaded programs. RLSOD considerably improves the precision of LSOD, while giving up on full scheduler independence (by restricting RLSOD to truly probabilistic schedulers). This same tradeoff has been proposed by earlier authors. Smith [5] improves on PN based on probabilistic bisimulation, where the latter forbids the execution time of any thread to depend on secret input. Just as in our work, a probabilistic scheduler is assumed; the probability of any execution step is given by a markov chain. A secure program requires that the probability to go from one low-equivalence class of states A to another (after possibly remaining, or *stuttering* in A for some time) is independent of the specific state $a \in A$. This approach is called *weak* probabilistic bisimulation, and allows the execution time of threads to depend on secret input, as long as it is not made observable by writing to public variables. The authors present a static check in form of a type system, and discuss an extension for thread creation. If the execution time up to the current point depends on secret input, their criterion allows to spawn new threads only if they do not alter public variables. In comparison, our c *cda* (n,m) based check *does* allow two public operations to happen in parallel in newly spawned threads, even if the execution time up to c (i.e.: a point at which at most one of the two threads involved existed) depends on secret input.

Approaches for PN based on type systems benefit from *compositionality*, a good study of which is given in [22]. Again, a probabilistic scheduler is assumed. Scheduler-independent approaches can be found in, e.g., [20, 35]. The authors each identify a natural class of "robust" resp. "noninterfering" schedulers, which include uniform and round-robin schedulers. They show that programs which satisfy specific possibilistic notions of bisimilarity ("FSI-security" resp. "possibilistically noninterferent") remain probabilistically secure when run under such schedulers. Since programs like Figure 7 left are not probabilistically secure under a round-robin scheduler, their possibilistic notion of bisimilarity require

$$\text{ASS}\ \frac{dom\,(exp) \sqsubseteq dom\,(var)}{\vdash \texttt{var} := \texttt{exp} : (dom\,(var),\ L)} \qquad \text{SKIP}\ \frac{}{\vdash \texttt{skip} : (H,\ L)}$$

$$\text{IF}\ \frac{\vdash com_1 : (ass,\ stp) \qquad \vdash com_2 : (ass,\ stp) \qquad dom\,(exp) \sqsubseteq ass}{\vdash \texttt{if}\ exp\ \texttt{then}\ com_1\ \texttt{else}\ com_2 : (ass,\ stp \sqcup dom\,(exp))} \qquad \text{SPAWN}\ \frac{\forall i \in \{0,\ldots,k-1\}\,.\, \vdash com_i : (ass,\ stp_i)}{\vdash \texttt{spawn}\,[com_0,\ldots,com_{k-1}] : (ass,\ L)}$$

$$\text{WHILE}\ \frac{\vdash com : (ass,\ stp) \qquad stp \sqcup dom\,(exp) \sqsubseteq ass}{\vdash \texttt{while}\ exp\ \texttt{do}\ com : (ass,\ stp \sqcup dom\,(exp))} \qquad \text{SUB}\ \frac{\vdash com : (ass',\ stp') \qquad ass \sqsubseteq ass' \qquad stp' \sqsubseteq stp}{\vdash com : (ass,\ stp)}$$

$$\text{SEQ}\ \frac{\vdash com_1 : (ass_1,\ stp_1) \qquad \vdash com_2 : (ass_2,\ stp_2) \qquad stp_1 \sqsubseteq ass_2}{\vdash com_1;\ com_2 : (ass_1 \sqcap ass_2,\ stp_1 \sqcup stp_2)}$$

Fig. 11. Security Type System FSI [20]

```
1  void main():
2    H := Hin;
3    if (H < 1234)
4       Lout := Lout + "0";
5    L := H;
6    Lout := Lout + str(L);
```

```
1    H := Hin;
2    if (H) {
3      skip };
4    spawn [
5      Lout := Lout + "17",
9      Lout := Lout + "42"
10   ]
```

Fig. 12. FSI-Variants of Figure 1, left, and of Figure 7, left

"lock-step" execution at least for threads with low-observable behaviour. Compared to RLSOD this is more restrictive for programs, but less restrictive on scheduling.

A completely different approach to noninterference and PN is the use of program logics and verification. For example the KeY system [36] has been used to verify noninterference and other nonfunctional security properties; KeY was also applied to Küster's e-voting system (cmp. chapter 7) [37]. Recently, relational program logics have been proposed which allow to express probabilistic properties of programs, including PN [38]. Such verification systems are very powerful and allow to verify properties beyond PN, but are never automatic such as RLSOD and JOANA. We are thus exploring the combination KeY + JOANA: JOANA can, due to a programming interface, export relevant analysis results, such as points-to relations, PDG reachability, PN guarantees, or leak localizations. KeY can use JOANA as a black box to considerably simplify security verification [39].

9.1. Detailed Comparison with Type-System Based Analysis for FSI

In Figure 11, the type system for FSI-Security from [20] for a concurrent WHILE language is repeated verbatim. FSI-Security implies $\mathcal{S}$-Security for all *robust* schedulers $\mathcal{S}$. Both these notions require an explicit global classification *dom* of all variables, implying a classification for all expressions. The attacker is assumed to observe the initial and final values of low variables. Intuitively, a judgment $\vdash com : (ass,\ stp)$ means that *com* is secure, with a running time influenced only by input of level stp or lower, and observable effects (specifically: writes to variables) only of level ass or higher.

The observational model in our work differs slightly: we assume the attacker to observe not the final value of low variables, but instead the values of variables at observable nodes, i.e.: nodes n with $ucl(n) = L$. The two FSI programs corresponding to Figure 1, left, and to Figure 7, left are shown in Figure 12, assuming $dom\,(\texttt{Hin}) = H, dom\,(\texttt{Lout}) = L$.

Comparing RLSOD with FSI, consider the example in Figure 12, left; which contains simple explicit and implicit leaks. RLSOD requires $cl(2) \sqsubseteq cl(3) \sqsubseteq cl(4)$ via the PDG path $2 \to 3 \to 4$ and requirement (a) in Definition 14, violating $H = ucl(2) = cl(2),\ L = ucl(4) \sqsupseteq cl(4)$ (via requirement (c)); and the leak is exposed. Likewise, FSI requires

$$dom\,(\texttt{H}) \overset{\text{IF}}{\sqsubseteq} ass_4, \quad \text{violating} \quad H = dom\,(\texttt{Hin}) \overset{\text{ASS}}{\sqsubseteq} dom\,(\texttt{H}), \quad ass_4 \overset{\text{ASS,SUB}}{\sqsubseteq} dom\,(\texttt{Lout}) = L.$$

where com_i is the statement at line i, and ass_i, stp_i are the security levels in judgements $\vdash com_i : (ass_i,\ stp_i)$ appearing in candidate derivation trees for the hypothesis $\vdash main : (ass,\ stp)$. $ass_{k_1,\ldots,k_n}$, $stp_{k_1,\ldots,k_n}$ are the security levels in judgements $\vdash com_{k_1,\ldots,k_n} : (ass_{k_1,\ldots,k_n},\ stp_{k_1,\ldots,k_n})$ for sequentially composed statements $com_{k_1,\ldots,k_n} = (\ldots(com_{k_1};\ com_{k_2});\ \ldots\ com_{k_n})$. Thus FSI discovers the same leak.

But FSI does not use the "dominator trick". To see the effect, consider Figure 12, right, which is secure w.r.t. a probabilistic scheduler, but FSI-insecure as follows: Aside from $H = dom\,(\texttt{Hin}) \overset{\text{ASS}}{\sqsubseteq} dom\,(\texttt{H})$, we have $dom\,(\texttt{H}) \overset{\text{IF}}{\sqsubseteq} stp_2$, and also $ass_5 \overset{\text{ASS,SUB}}{\sqsubseteq} dom\,(\texttt{Lout}) = L$. But attempting to derive a type for $com_{2,4} = com_2;\ \texttt{spawn}\,[\texttt{Lout := Lout + "17",}\ldots]$ via rule SEQ requires $stp_2 \sqsubseteq ass_4 \overset{\text{SPAWN,SUB}}{\sqsubseteq} ass_5$. The latter is a contradiction, thus FSI is violated.

RLSOD, however, correctly classifies the corresponding program from Figure 7 left secure w.r.t. a probabilistic scheduler, as witnessed by the classification

line	2	3	4	5	9
cl	H	H	L	L	L

Specifically, this classification does *not* violate requirement (b) of Definition 14, since $4\ cda\ (5, 9)$, but for all nodes c' on paths $4 \to 5, 4 \to 9$ we have $cl(c') = L$. The dominator avoids a false alarm.

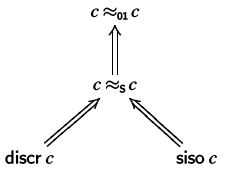
9.2. Comparison with Syntactic Criteria for Resumption Based Noninterference

In [22], the authors assume a uniform scheduler, and describe a lattice of security properties (Figure 13) ordered by implication, the weakest of which being 01-bisimilarity ($\approx_{01}$). 01-bisimilarity is not compositional w.r.t. every syntactic construct of the source language considered. Nevertheless, the authors derive syntactic criteria that, whenever a given construct is not compositional w.r.t. the security property, defer to a stronger property which *is* compositional w.r.t. this construct.

Instead of $\texttt{spawn}\,[com_0, \ldots, com_{k-1}]$ or `fork`, their language assumes a parallel composition statement $\texttt{par}\,[com_0, \ldots, com_{k-1}]$. For a direct comparison, the program corresponding to Figure 7 left is shown in Figure 13. Each line is annotated with its strongest security properties holding. For example, `read(H)` is self-isomorphic (siso): if started in low-indistinguishable states (and: given low-indistinguishable inputs), executions take the same branches with the same probabilities. It also is discreet (discr): during the computation, the states stay low-indistinguishable from the initial state, and no low output is made. Using the syntactic criteria, we can conclude that `read(H); if (H) {skip}` is discr (but not: siso). Unfortunately, the syntactic criteria of [22] do not allow us to conclude that sequential composition of a discr and a siso command (like the `par` []-Statement) are $\approx_{01}$, and hence produce a false alarm.

9.3. Compositionality

Compositionality is a useful property, but can be achieved only at the price of either losing precision, or losing automatic analysis. Let us explain this in more detail.




```
1    read(H);          siso, discr
2    if (H) {skip};    discr
3    par [
4       print(17),     siso
5       print(42)      siso
6    ];                siso
```

Fig. 13. Resumption Based Security: Security Properties[22], and the Program corresponding to Figure 7, left

```
1  void main():
2    read(H);
3    T = H;
4
5    L = T;
6    print(L);
```

```
1  void main():
2    read(H);
3    T = H;
4    ...
5    T = 0;
6    print(T);
```

```
1  void main():
2    read(H);
3    if (H > 0) {
4       H--;
5    }
6    print(0);
```

Fig. 14. Typical problems of compositional analyses.

Typically, type systems are compositional because they use static classification. For example, in figure 14 left, the first two lines alone are secure, as well as the last two. A compositional IFC with static classification will discover the simple leak in this code piece once both fragments are composed, because there are conflicting (static) classifications for `T`. Unfortunately, static classification is very unprecise. In particular, it is not flow sensitive and will produce a false alarm for the program in figure 14 middle: since the high value is overwritten in line 4, the program always outputs 0 and is secure. Note that there might be many more statements between the two writes to `T` (cmp. [12]).

For concurrent programs, compositionality is even more problematic. Consider the program in figure 14 right: it is sequential and always prints 0, and therefore fulfills PN for every scheduler. If this program is composed in parallel with the secure program `print(1);`, it however contains a probabilistic leak similar to figure 1 right. Therefore, a typical compositional analysis (e.g. [5, 20]) will reject the isolated code piece in figure 14 right, because it conservatively assumes that it can be composed with arbitrary (unknown) threads. RLSOD in contrast is a whole-program analysis, and uses MHP information to exactly infer possible concurrency (cmp. section 4.1). Thus it considers the isolated code piece secure – which it is. It should be noted that compromises between the two extreme positions "fully compositional" vs. "whole-program" have been attempted, e.g. [40, 41].

Program logics as discussed above are also compositional, typically because they use Hoare triples. In contrast to type systems and RLSOD, program logics for PN can be made arbitrarily precise. But the price is high: program logics and verification are never fully automatic, and in practice require high manual effort for specification and verification. We thus believe that in practice, RLSOD is a good compromise: it is fully automatic yet very precise, and compositionality can be simulated by stubs.

10. Conclusion

We described a new algorithm for probabilistic noninterference, named RLSOD, which allows secure low-nondeterminism, while basically maintaining the low-deterministic security (LSOD) approach. RLSOD benefits from flow- and context-sensitive program analysis methods such as PDGs, points-to analysis, and dominators in multi-threaded programs. It turns out that RLSOD heavily reduces false alarms

compared to LSOD, while a full soundness proof could be achieved. An Isabelle formalization of the soundness proof has already begun.

RLSOD is integrated into the JOANA IFC tool. JOANA can handle full Java with arbitrary threads, while being sound and scaling to 200k LOC. The decision to base PN in JOANA on low-deterministic security was made at a time when mainstream IFC research considered LSOD too restrictive. In the current paper we have shown that flow- and context-sensitive analysis, together with new techniques for allowing secure low-nondeterminism, has rehabilitated the LSOD idea.

References

[1] D. Giffhorn and G. Snelting, A new algorithm for low-deterministic security, *International Journal of Information Security* **14**(3) (2015), 263–287.

[2] J. Breitner, J. Graf, M. Hecker, M. Mohr and G. Snelting, On improvements of low-deterministic security, in: *Proc. Principles of Security and Trust (post 2016)*, Lecture Notes in Computer Science, Vol. 9635, Springer Berlin Heidelberg, 2016, pp. 68–88.

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

[4] G. Smith and D. Volpano, Secure information flow in a multi-threaded imperative language, in: *Proc. Popl '98*, ACM, 1998, pp. 355–364.

[5] G. Smith, Improved typings for probabilistic noninterference in a multi-threaded language, *J. Comput. Secur.* **14**(6) (2006), 591–623.

[6] A. Sabelfeld and D. Sands, Probabilistic noninterference for multi-threaded programs, in: *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, Uk, July 3-5, 2000*, 2000, pp. 200–214.

[7] A.W. Roscoe, J. Woodcock and L. Wulf, Non-interference through determinism, in: *Esorics*, Lncs, Vol. 875, 1994, pp. 33–53.

[8] S. Zdancewic and A.C. Myers, Observational determinism for concurrent program security, in: *Proc. Csfw*, IEEE, 2003, pp. 29–43.

[9] M. Huisman, P. Worah and K. Sunesen, A temporal logic characterisation of observational determinism, in: *Proc. 19th Csfw*, IEEE, 2006, p. 3.

[10] M. Huisman and T.M. Ngo, Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification, in: *Proc. Formal Verification of Object-oriented Systems*, 2011.

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

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

[13] M. Mohr, J. Graf and M. Hecker, JoDroid: Adding Android support to a static information flow control tool, in: *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. März 2015*, Ceur Workshop Proceedings, Vol. 1337, CEUR-WS.org, 2015, pp. 140–145.

[14] R. Küsters, E. Scapin, T. Truderung and J. Graf, Extending and applying a framework for the cryptographic verification of Java programs, in: *Proc. Post 2014*, Lncs 8424, Springer, 2014, pp. 220–239.

[15] R. Küsters, T. Truderung and J. Graf, A framework for the cryptographic verification of Java-like programs, in: *Computer Security Foundations Symposium (csf), 2012 Ieee 25th*, IEEE Computer Society, 2012.

[16] J. Graf, M. Hecker, M. Mohr and G. Snelting, Checking applications using security APIs with JOANA, 2015, 8th International Workshop on Analysis of Security APIs.

[17] J. Graf, M. Hecker, M. Mohr and G. Snelting, Tool demonstration: Joana, in: *Proc. Principles of Security and Trust (post 2016)*, Lecture Notes in Computer Science, Vol. 9635, Springer Berlin Heidelberg, 2016, pp. 89–93.

[18] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr and D. Wasserrab, Checking probabilistic noninterference using JOANA, *It – Information Technology* **56** (2014), 280–287.

[19] D. Wasserrab, D. Lohner and G. Snelting, On PDG-based noninterference and its modular proof, in: *Proc. Plas '09*, ACM, 2009.

[20] H. Mantel and H. Sudbrock, Flexible scheduler-independent security, in: *Computer Security – Esorics 2010*, D. Gritzalis, B. Preneel and M. Theoharidou, eds, Lecture Notes in Computer Science, Vol. 6345, Springer Berlin Heidelberg, 2010, pp. 116–133. ISBN 978-3-642-15496-6.

[21] A. Askarov, S. Hunt, A. Sabelfeld and D. Sands, Termination-insensitive noninterference leaks more than just a bit, in: *Proc. Esorics*, Lncs, Vol. 5283, 2008, pp. 333–348.

[22] A. Popescu, J. Hölzl and T. Nipkow, Formalizing probabilistic noninterference, in: *Certified Programs and Proofs – Third International Conference, CPP 2013, Melbourne, Vic, Australia, December 11-13, 2013, Proceedings*, 2013, pp. 259–275.
[23] S. Hunt and D. Sands, On flow-sensitive security types, in: *Popl '06*, ACM, 2006, pp. 79–90.
[24] C. Hammer, Experiences with PDG-based IFC, in: *Proc. Essos'10*, F. Massacci, D. Wallach and N. Zannone, eds, Lncs, Vol. 5965, Springer-Verlag, 2010, pp. 44–60.
[25] T.M. Ngo, Qualitative and quantitative information flow analysis for multi-threaded programs, PhD thesis, University of Enschede, 2014.
[26] G. Snelting, Combining slicing and constraint solving for validation of measurement software, in: *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, Springer-Verlag, London, UK, 1996, pp. 332–348.
[27] G. Snelting, T. Robschink and J. Krinke, Efficient path conditions in dependence graphs for software safety analysis, *Acm Trans. Softw. Eng. Methodol.* **15**(4) (2006), 410–457.
[28] S. Horwitz, J. Prins and T. Reps, On the adequacy of program dependence graphs for representing programs, in: *Proc. Popl '88*, ACM, New York, NY, USA, 1988, pp. 146–157.
[29] T. Reps, S. Horwitz, M. Sagiv and G. Rosay, Speeding up slicing, in: *Proc. Fse '94*, ACM, New York, NY, USA, 1994, pp. 11–20.
[30] D. Giffhorn, Advanced chopping of sequential and concurrent programs, *Software Quality Journal* **19**(2) (2011), 239–294.
[31] G. Naumovich and G.S. Avrunin, A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel, in: *Proceedings of the 6th Acm Sigsoft International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, 1998, pp. 24–34.
[32] L. Li and C. Verbrugge, A practical MHP information analysis for concurrent Java programs, in: *Proc. Lcpc'04*, Lncs, Vol. 3602, Springer, 2004, pp. 194–208.
[33] J. 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, 2016.
[34] B. De Sutter, L. Van Put and K. De Bosschere, A practical interprocedural dominance algorithm, *Acm Trans. Program. Lang. Syst.* **29**(4) (2007).
[35] A. Popescu, J. Hölzl and T. Nipkow, Noninterfering schedulers, in: *Algebra and Coalgebra in Computer Science*, R. Heckel and S. Milius, eds, Lecture Notes in Computer Science, Vol. 8089, Springer Berlin Heidelberg, 2013, pp. 236–252.
[36] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P.H. Schmitt and M. Ulbrich (eds), *Deductive Software Verification - The Key Book - From Theory to Practice*, Lecture Notes in Computer Science, Vol. 10001, Springer, 2016.
[37] R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten and M. Mohr, A hybrid approach for proving noninterference of java programs, in: *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 305–319.
[38] G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy and S.Z. Béguelin, Probabilistic relational verification for cryptographic implementations, in: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, Ca, Usa, January 20-21, 2014*, 2014, pp. 193–206.
[39] S. Greiner, M. Mohr and B. Beckert, Modular verification of information flow security in component-based systems, in: *Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings*, 2017, pp. 300–315.
[40] H. Mantel, M. Müller-Olm, M. Perner and A. Wenner, Using dynamic pushdown networks to automate a modular information-flow analysis, in: *Pre-proceedings of the 25th International Symposium on Logic Based Program Synthesis and Transformation (lopstr)*, 2015.
[41] H. Mantel, D. Sands and H. Sudbrock, Assumptions and guarantees for compositional noninterference, in: *Proceedings of the 24th Ieee Computer Security Foundations Symposium (csf)*, IEEE Computer Society, Cernay-la-Ville, France, 2011, pp. 218–232.