Low-Deterministic Security For Low-Nondeterministic Programs

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 both: 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

---

1A 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”.

0926-227X/17/$35.00 © 2017 – IOS Press and the authors. All rights reserved
Snelting et al. / Low-Deterministic Security For Low-Nondeterministic Programs

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 kills LSOD from a practical viewpoint.

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 maintaining the LSOD approach. It requires – like many other algorithms, e.g. [4, 5] – that the scheduler is probabilistic. The algorithm was described in detail in [1, 12]; it is integrated into the JOANA IFC tool.

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) 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, and show how RLSOD avoids false alarms.

Our work builds heavily on our earlier contributions [1, 12], but the current article is aimed to be self-contained.

2. Noninterference and LSOD

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

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.

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].

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”.

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 ⊥ = L ⩽ H = ⊤ lattice. Note also that integrity is dual to confidentiality, but will not be discussed here. JOANA can handle both.
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;
10        while (H != 0)
11            H--;
12        print("CS");
13
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.

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”.

2.1. Sequential Noninterference

To formalize RLSOD, let us start with 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. Note that flow-sensitive IFC such as RLSOD does not use a static, global classification of variables; this will be explained in section 3.1.

Definition 1 (Sequential noninterference). Let \( P \) be a program. Let \( s, s' \) be initial program states, let \( \llbracket P \rrbracket(s), \llbracket P \rrbracket(s') \) be the final states after executing \( P \) in state \( s \) resp. \( s' \). Noninterference holds iff

\[
s \sim_L s' \implies \llbracket P \rrbracket(s) \sim_L \llbracket P \rrbracket(s').
\]

The relation \( s \sim_L s' \) means that two states are low-equivalent, that is, coincide on low variables: \( cl(v) = L \implies s(v) = s'(v) \). 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. Truly interactive programs lead to the problem of termination leaks [19], which will not be explored in this paper.

2.2. Probabilistic Noninterference

In multi-threaded programs, fine-grained interleaving effects must be accounted for, thus traces are used instead of states. A trace is a sequence of events \( t = (s_1, o_1, s_2), (s_2, o_2, s_3), \ldots, (s_r, o_r, s_{r+1}), \ldots \), where the \( o_r \) are operations (i.e. dynamically executed program statements \( c_r \); we write \( stmt(o_r) = c_r \)). Operations are unique within a trace. \( s_r, s_{r+1} \) are the states before resp. after executing \( o_r \). We write \( e \)
As certain inputs. Thus we define why flow-sensitive IFC is more precise (cmp. [1], sec. 3). More traces to be low-equivalent without compromising soundness. This subtle observation is another (1) For an operation Definition 2. Then includes filtering out high operations from traces. This leads to (2) The low-observable part of an event is defined as

\[ E_L((s, o, s')) = \begin{cases} \{s|_{\text{use}(o)}, o, s'|_{\text{def}(o)}\}, & \text{if } cl(o) = L \\ \epsilon, & \text{otherwise} \end{cases} \]

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

\[ LS(t) = \text{map}(E_L)(\text{filter}(\lambda e. E_L(e) \neq \epsilon)(t)). \]

(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 public behaviour. Thus \( [t]_L \) represents \( t \)'s low behaviour. Note also that the flow-sensitive projections \( s|_{\text{def}}(o) \), \( s|_{\text{use}}(o) \) 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 3. (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 is classical, and uses explicit input streams instead of initial states. For both inputs the same initial state is assumed, but it is assumed that all input values are classified low or high. Input streams \( i = i_1i_2 \ldots, t' = t'_1t'_2 \ldots \) are low equivalent \( (i \sim_L t') \) if they coincide on low values:

\[ cl(i_s) = L \land cl(t'_s) = L \implies i_s = t'_s. \]

The definition relies on our flow-sensitive \( t \sim_L t' \).
Definition 4 (Probabilistic noninterference). Let $i, i'$ be input streams; let $T(i)$ be the set of all possible traces of program $P$ for input $i$, $\Theta = T(i) \cup T(i')$. PN holds iff

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

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

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

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")...
...print("J")...$\in t$, or...

...print("CS")...
...print("J")...$\in t$. There are no other low events or low output, hence there are only two equivalence classes

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

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

Now if $i$ contains a small value, $i'$ a large value, as discussed earlier $P_i([t]_L^1) \neq P_{i'}([t]_L^1)$ as well as $P_i([t]_L^2) \neq P_{i'}([t]_L^2)$, 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, 20]. Fortunately, explicit probabilities are not needed for soundness proofs. We further assume that statements and operations are deterministic; nondeterminism can only result from scheduling. 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 $t \in T(i)$ we have $P_i(t) = 1$. 

---

Only RLSOD avoids false alarm. Right: only RLSOD avoids false alarm.

Fig. 2. Left: insecure program, obvious explicit leak. Middle: secure program, Giffhorn's criterion + flow sensitivity avoid false trace of program $P_i$ executed is the same under $i$.

Now if $i$ have been used to explicitly determine the $P_i$ for sequential programs $PN$ implies sequential noninterference. Note that for sequential (deterministic) programs $|T(i)| = 1$, and for the unique $t \in T(i)$ we have $P_i(t) = 1$. 

---

```
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);
11 void thread_2():
12    print(L);
13    L = H;
14```
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 \( t' \) 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 \( [P](s) \sim_L [P](s') \). □

2.3. 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 strictly deterministic. This is however a very restrictive requirement and eventually led to popular scepticism against LSOD.

Definition 5 (Low-security observational determinism). Let \( i, i', \Theta \) as above. LSOD holds iff

\[
i \sim_L i' \iff \forall t, t' \in \Theta : t \sim_L t'.
\]

Under LSOD, all traces \( t \) for input \( i \) are low-equivalent: \( \forall t' \in T(i) : t' \sim_L t \), thus \( T(i) \subseteq [i]_L \). 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 \( T(i) \subseteq [i]_L \). As \( P_i(t') = 0 \) for \( t' \notin T(i) \), we have

\[
P_i([t]_L) = \sum_{t' \in [i]_L} P_i(t') = \sum_{t' \in T(i)} P_i(t') = 1
\]

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 2. In Figure 2 middle, statements \( \text{print}(L) \) 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.
3. 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.

3.1. PDGs for IFC

Snelting proposed to use Program Dependence Graphs (PDGs) as a device to check integrity of software as early as 1995 [21]. Later the approach was expanded into the JOANA IFC project. It was shown that PDGs guarantee sequential noninterference [22], 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 2 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 [23] □

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, 24, 25].

As an example, consider Figure 3. The left PDG has a data dependency edge from \( L=H; \) to \( \text{print}(L) \); because \( L \) is defined in line 9 (Figure 2 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 \( \text{read}(H) \); to \( \text{print}(L) \);, 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 \( \text{print}(L) \); due to flow sensitivity: no scheduler
will ever execute \( L = H; \) before \( \)\( \text{print}(L); \) . Hence no path from \( \text{read}(H) \) to \( \text{print}(L); \) 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, 26, 27]. 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 6.** 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

\[
\text{BS}(n) = \{ m | m \rightarrow_R^* n \}
\]

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

We will now formally define the propagation of user (“engineer”) classifications in the PDG. This definition will be expanded later to cover concurrent programs.

**Definition 7.**

1. Let \( \mathcal{L} \) be the lattice of security levels (in this paper we use \( \mathcal{L} = \{H, L\} \) where \( L \leq H \)). Let \( I \subseteq N \) be the set of input nodes (“sources”) and \( O \subseteq N \) be the set of output nodes (“sinks”) in the PDG. The engineer has to provide user-defined classifications \( ucl : I \cup O \rightarrow \mathcal{L} \).

2. Other PDG nodes need not be explicitly classified, but a classification can be computed via the flow equation

\[
cl(n) = \bigcup_{m \rightarrow n} cl(m)
\]

with additional constraints \( \forall n \in I : ucl(n) \leq cl(n) \) and \( \forall n \in O : ucl(o) \geq cl(o) \).

3. For an operation \( o \) in a trace \( t \), we assume \( \text{stmt}(o) \in N \) and define \( cl(o) = cl(\text{stmt}(o)) \).

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 occurrence 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 \( cl(n) = \bigcup_{m \rightarrow n} cl(m) \). The latter can easily be computed by 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, 28].

The soundness theorem for PDG-based sequential IFC can now be formalized:
Theorem 2. Sequential noninterference holds if
\[ \forall n, n' \in N : \text{cl}(n') = L \land \text{cl}(n) = H \Rightarrow n \notin BS(n') \]

Proof. Snelting’s original proof was in [22]; additional details are given in [12]. □

3.2. 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 original 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 (see [1] for details). Giffhorn provided a new definition for low-equivalent traces:

Definition 8 (cf. [1]). 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 2);
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 9. (1) We write \( \text{MHP}(n, m) \) if MHP analysis concludes that \( n \) and \( m \) may be executed in parallel. That is, if traces \( t, t' \) exist where
\[
\begin{align*}
t &= \ldots (s_\nu, o_\nu, s_\nu+1) \ldots (s_\mu, o_\mu, s_\mu+1) \ldots, \\
t' &= \ldots (s'_\nu, o'_\nu, s'_\nu+1) \ldots (s'_\mu, o'_\mu, s'_\mu+1) \ldots,
\end{align*}
\]
\[ n = \text{stmt}(o_\nu), \quad m = \text{stmt}(o_\mu), \quad \text{then} \ \text{MHP}(n, m), \]

2. \( \text{Ind}(n, n') \iff \text{MHP}(n, n') \land \text{cl}(n) = \text{cl}(n') = L \), which denotes that \( n, n' \) are low-nondeterministic;

3. \( \text{race}(n, n') \iff \text{MHP}(n, n') \land \exists v \in \left( \text{def}(n) \cap (\text{def}(n') \cup \text{use}(n')) \right) \cup \left( \text{def}(n') \cap (\text{use}(n) \cup \text{def}(n)) \right) \), which denotes there is a data race between \( n, n' \);

4. \( \text{path}(n, n') \iff n \rightarrow^*_{\text{CFG}} n' \) is a path in the CFG. We will also use \( \text{path}(n, n') \) 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:
Fig. 4. Violations of condition 2 from definition 10. Top: $n = L = 1, n' = H = H + L, n'' = \text{print}(X), n \in BS(n'')$. Bottom: $n = H = 42, n' = H - -, n'' = \text{print}(L), n' \in BS(n'')$.

**Definition 10.** The PDG-based LSOD criterion is satisfied if

1. $\forall n, n' \in N: cl(n) = L \land cl(n') = H \implies n' \notin BS(n)$,
2. $\forall n, n', n'' \in N: race(n, n') \land cl(n'') = L \implies n \notin BS(n'') \land n' \notin BS(n''')$,
3. $\forall n, n' \in N: \neg lnd(n, n')$.

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 conditions 2 and 3 are not completely disjoint, in particular if $cl(n) = cl(n') = cl(n'') = L, n \in BS(n'')$ both conditions are violated. In Figure 1 right, condition 3 is violated, while for the examples in Figure 4, condition 3 is not violated, but condition 2 is violated.

The PDG-based Zdancewic criterion assumes that the attacker can see all low operations. While this seems appropriate for classic noninterference, it is questionable in a flow-sensitive setting: remember that a variable may have several PDG occurrences with varying classification, and that the engineer explicitly specifies only the classification of sources and sinks, while the internal classification according to definition 7 serves only to ensure soundness. Thus Giffhorn eventually decided that in his attacker model, only low sources and sinks are low observable – giving the engineer full control of low observability via the $ucl$ definition. Consequently, Giffhorn used a more realistic definition of low-equivalent inputs: while the classical definition assumes one input stream with input values of varying classification, Giffhorn assumes several input streams with fixed classification each. This leads to

**Definition 11.** (1) The classification of input stream $S$ which is part of program input $i$, and read at input statement $n \in I$ is $cl(S) = cl(n)$. If several $n \in I$ read $S$, we assume they have the same classification. The low input stream is written $S_L$.

(2) Inputs are considered low equivalent if their low input streams are equal: $i \sim_L i' \iff S'_L = S''_L$.

The traditional definition of low-equivalence is still useful in proofs. To distinguish it from Giffhorn’s low-equivalence, from now on we call it $cl$-equivalence and use $cl$ as a subscript instead of $L$. e.g. as in $[t]_{cl}$ or $t \sim_{cl} t'$. This also emphasizes the fact that it depends on the classification $cl$. 
Furthermore, Giffhorn discovered that only two of the three classical race situations ("write-write, write-read, read-write") are relevant, because the case \( v \in \text{def}(n) \cap \text{use}(n') \), \( n \in \text{BS}(n'') \) can never cause a leak! He thus simplified the definition of a race, introducing an “asymmetrical” race notion:

\[ \text{race}(n,n') \iff \text{MHP}(n,n') \land \exists v \in (\text{def}(n) \cap (\text{def}(n') \cup \text{use}(n'))) \]

Thus the example from the top of Figure 4 is no longer considered insecure. Giffhorn’s two remarkable insights lead to a slightly modified (weaker) criterion, which is potentially more precise:

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

1. \( \forall n \in O, n' \in I: \text{ucl}(n) = L \land \text{ucl}(n') = H \implies n' \notin \text{BS}(n) \),
2. \( \forall n, n' \in N; n'' \in O: \text{race}(n,n') \land \text{ucl}(n'') = L \implies n' \notin \text{BS}(n'') \),
3. \( \forall n, n' \in I \cup O: \neg \text{ind}(n,n') \).

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

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

Note that definition 10 implies definition 12, as conditions 1. – 3. are demanded for all \( n \in N \), and \( \text{ucl} \) satisfies the conditions in definition 6 (item 3).

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. In Figure 2 left, a leak is discovered according to condition 1, which is also correct (cmp. PDG example above). In Figure 2 middle and right, the explicit leak has disappeared (thanks to flow-sensitivity), but another leak is discovered by condition 3: we have \( \text{MHP}(L = 42; \text{print}(L)); \), 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 \( \text{cl}(n'') = H \), to \( n \) or \( n' \), where \( \text{ind}(n, n') \). 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. 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 10 by

\[ \text{3}. \forall n, n' \in N: \exists n'' \in N: \text{cl}(n'') = H \land (\text{path}(n'', n) \lor \text{path}(n'', n')) \implies \neg \text{ind}(n, n') \]

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

\[ \text{3}. \forall n, n' \in N: \text{ind}(n, n') \implies \forall n'' \in \text{path}(\text{START}, n) \cup \text{path}(\text{START}, n'): \text{cl}(n'') = L \]

where \( \text{START} \) is the CFG’s Start node. The more precise “star” version (definition 12) reads

\[ \text{3}. \forall n, n' \in I \cup O: \text{ind}(n, n') \implies \forall n'' \in \text{path}(\text{START}, n) \cup \text{path}(\text{START}, n'): \text{cl}(n'') = L \]

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

\[ \text{3}. \forall n, n' \in N: \exists n'' \in \text{path}(\text{START}, n) \cup \text{path}(\text{START}, n'): \text{cl}(n'') = L \]

In [11] these races were called “data conflicts”.

\[ \text{In [11] these races were called “data conflicts”}.\]
2'. \( \forall n, n', n'' \in N: \text{race}(n, n') \land cl(n'') = L \land \exists n'' \in N: cl(n'') = H \land (\text{path}(n'', n) \lor \text{path}(n'', n')) \)
\[ \implies n, n' \notin BS(n''). \]

By contraposition, we obtain the more practical form

2'. \( \forall n, n', n'' \in N: \text{race}(n, n') \land cl(n'') = L \land (n \in BS(n'') \lor n' \in BS(n'')) \)
\[ \implies \forall n'' \in (\text{path}(\text{START}, n) \cup \text{path}(\text{START}, n')): cl(n'') = L \]

The more precise “star” version reads

2**. \( \forall n, n' \in N, n'' \in O: \text{race}(n, n') \land ucl(n'') = L \land n' \in BS(n'') \)
\[ \implies \forall n'' \in \text{path}(\text{START}, n) \cup \text{path}(\text{START}, n'): cl(n'') = L \]

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 2 middle is ok according to Giffhorn, because the low-nondeterminism in line 6 resp. 9 cannot be reached from any high statement (condition 3'). The same holds for the data race between line 6 and line 9 – condition 2' is violated (note that in this example, \( n' = n'' \)), but 2** holds. Indeed the program is PN. Figure 2 right is however not covered by Giffhorn’s criterion, because the initial \( \text{read} (\text{H}2) \) will reach any other statement. But the program is PN, because \( \text{H}2 \) does not influence any later low statement!

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. We will not present a formal soundness argument, as Giffhorn’s algorithm is a special case of RLSOD, which will be discussed in the next section.

4. 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 2 right (program \( \mathcal{P}_2 \)). 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 low-nondeterministic statements, 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 low 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
\[ [t]_L^1 = \{ t' | \ldots t' = \text{print("J")} \ldots \text{print("CS")} \ldots \} \]
\[ [t]_L^2 = \{ t' | \ldots t' = \text{print("CS")} \ldots \text{print("J")} \ldots \} \]
Likewise, \( \mathcal{P}_2 \) has two low classes
\[ [t]_L^1 = \{ t' | \ldots t' = \text{L = 42} \ldots \text{print(42)} \ldots \} \]
\[ [t]_L^2 = \{ t' | \ldots t' = \text{print(0)} \ldots \text{L = 42} \ldots \} \]

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 \( \mathcal{P}_2 \), \( P_{t_i}(i)_{L_2} = P_{t_i}(i)_{L_1} \) holds!
Technically, $P_2$ contains a point $c$ which dominates both low-nondeterministic statements $n \equiv L = 42$; and $m \equiv \text{print}(L)$, and all relevant high events always happen before $c$. Domination means that any control flow from $\text{START}$ to $n$ or $m$ must pass through $c$. In $P_2$, $c$ is the point immediately before the first fork. In contrast, $P_1$ has only a trivial common dominator for the low nondeterminism, namely the $\text{START}$ node, and on the path from $\text{START}$ to $n \equiv \text{print}("J")$ there is no high event, while on the path to $m \equiv \text{print}("CS")$ 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 $P_2$ contains an additional high statement $m' \equiv \text{read}(H)$ but that is behind $n$ (no control flow is possible from $m'$ to $n$) and thus cannot influence the $n/m$ nondeterminism.

4.1. 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 6). We will now provide precise definitions for this idea.

**Definition 13** (Common dynamic ancestor). Let $n, m \in \mathcal{P}$ be statements.

1. Statement $c$ is a dominator for $n$, written $c \text{ dom } n$, if $c$ occurs on every CFG path from $\text{START}$ to $n$.
2. Statement $c$ is a common dominator for $n, m$, written $c \text{ cdom } (n, m)$, if $c \text{ dom } n \land c \text{ dom } m$.
3. Statement $c$ is a common dynamic ancestor for $n, m$, written $c \text{ cda } (n, m)$, if $c \text{ cdom } (n, m) \land \neg \text{MHP}(c, n) \land \neg \text{MHP}(c, m)$.
4. If $c \text{ cda } (n, m)$ and $\forall c' \text{ cda } (n, m)$: $c'$ 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 6). 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. We are now ready to formally define the improved RLSOD criterion.

Definition 14 (RLSOD). RLSOD holds if LSOD condition 1 holds for all PDG nodes, and if

3. ∀n, n' ∈ N: ind(n, n') ∧ c = cda(n, n') → ∀n'' ∈ path(c, n) ∪ path(c, n'): cl(n'') = L
2. ∀n, n', n'' ∈ N: race(n, n') ∧ c = cda(n, n') ∧ cl(n'') = L ∧ (n ∈ BS(n'') ∨ n' ∈ BS(n''))
   → ∀n''' ∈ path(c, n) ∪ path(c, n'): cl(n''') = L

The “star” version reads

3*. ∀n, n' : ind(n, n') ∧ c = cda(n, n') → ∀n'' ∈ path(c, n) ∪ path(c, n') : cl(n'') = L
2*. ∀n, n' ∈ N, n'' ∈ O: race(n, n'') ∧ c = cda(n, n') ∧ cl(n'') = L ∧ n' ∈ BS(n'')
   → ∀n''' ∈ path(c, n) ∪ path(c, n') : cl(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 dominator. Figure 5 illustrates the RLSOD definition (condition 3*). Note that Giffhorn’s original criterion trivially fulfils conditions 2* and 3*, where cda always returns START. Thus RLSOD is a true generalization of Giffhorn’s algorithm.

4.2. Classification Revisited

Consider the program in Figure 6 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 line 1 or line 2 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 reached from the high statement in line 3 (middle). Applying RLSOD, the common dynamic ancestor for the two print statements is line 10.

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

But according to the flow equation, 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! This is not a flaw in condition 3**, but an incompleteness in the flow equation – it must be extended for nondeterminism.

---

Note that in programs with procedures and threads, immediate dynamic ancestors may not be unique due to context-sensitivity [29].
thread1() {
  if (H) {
    skip;
  }
  fork Thread2();
  print(17);
}

thread2() {
  print(42);
}

thread1() {
  tmp = 1;
  if (H) {
    tmp = 100;
  }
  fork thread2();
  while (tmp > 0) {
    tmp = tmp - 1;
  }
  tmp2 = 1;
  fork thread3();
  while (tmp2 > 0) {
    tmp2 = tmp2 - 1;
  }
  print(L1);
}

thread2() {
  tmp2 = 100;
}

thread3() {
  print(L2);
}

Fig. 6. Left: deterministic round-robin scheduling may leak. Middle/Right: a leak which goes undiscovered if classification of statements is incomplete.

In general, the rule is as follows. The standard flow equation \( cl(n) = \bigsqcup_{m \rightarrow n} cl(m) \) 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 \( \text{MHP}(n,m) \), and \( c = \text{cda}(n,m) \), and the path \( c \rightarrow^{*}_{CFG} n \) 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 15 (Classification in PDGs).** A PDG \( G = (N, \rightarrow) \) is classified correctly, if

(a) \( \forall n \in N : cl(n) \geq \bigsqcup_{m \rightarrow n} cl(m) \),
(b) \( \forall n, m \in N : \text{MHP}(n,m) \land c = \text{cda}(n,m) \land \exists c' \in \text{path}(c,n), cl(c') = H \implies cl(n) = H. \)
(c) \( \forall n \in I : ucl(n) = cl(n) \) and \( \forall n \in O : ucl(n) \geq cl(n) \).

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

In Figure 6 middle/right, definition 15 enforces line 10 to be classified high, as we have \( \text{cda}(10, 18) = 6 \), 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 5 will demonstrate that classifiability according to definition 15 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 15, 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 a 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 $n \rightarrow^* n'$. But then repeated application of rule (a) demands $cl(n') = H$, which is a contradiction.

(2) 3\* is violated: For the two $ind$ 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 15 under the notion of RLSOD. To actually use it as a PN checker, it uses a fixpoint iteration similar to the sequential one (definition 7, see also [12, 28]).

4.3. Scheduler Assumptions

Before we discuss soundness, let us point out an assumption which is standard for PN, namely that the scheduler is truly probabilistic. In particular, it maintains no state of its own, does not look at program variables, and the relative chance of two threads to be scheduled next is independent of other possibly running threads. The necessity of this assumption was stressed by various authors, e.g. [5]. Indeed a malicious scheduler can read high values to construct an explicit flow by scheduling, as in

\[
{H=0; \mid \mid H=1;} {L=0; \mid \mid L=1;}
\]

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 6 left. 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 $9 \rightarrow 5$ resp. $5 \rightarrow 9$, leaking H. However under the assumption of truly probabilistic scheduling, Figure 6 left is RLSOD.

We also assume sequential consistency. Under the Java Memory Model, for programs that are not correctly synchronized, the compiler may, e.g. through reordering, produce code that is not sequentially consistent. This limits static analysis considerably, so we ignore this possibility. Note that for correctly synchronized programs, the compiler must guarantee sequential consistency.

5. 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. $|\{(n, n') \mid ind(n, n') \lor race(n, n')\}| \leq 1$).
The full proof posed more difficulties than expected, but eventually led to a simpler formulation of the RLSOD criterion, namely in form of Definition 15. 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 15. As in [2], the proof relies on the notion of conditional probability for traces.

**Definition 16.** Let \( t_1 \cdots \) be the set of traces beginning with prefix \( t_1 \), so that \( P_i(t_1 \cdots) = \sum_{t \sim t_1} P_i(t) \) 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

\[
P_i(t_2 \mid t_1) = \frac{P_i(t_1 \cdot t_2)}{P_i(t_1 \cdots)}
\]

This notion extends to sets of traces:

\[
P_i(T' \mid T) = \frac{P_i(T \cdot T')}{P_i(T \cdots)} = \sum_{i \in T \cdot T'} P_i(t) / \sum_{i \in T \cdots} P_i(t)
\]

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

**Lemma 3.** (1) \( [t_1]_{c,d}[t_2]_{c,d} = [t_1 t_2]_{c,d} \)
(2) \( P_i([t]_{c,d}) = P_i([\varepsilon]_{c,d} \mid [t]_{c,d}) \cdot P_i([t]_{c,d} \cdot \cdots \mid [\varepsilon]_{c,d}) \)
(3) \( P_i([t_1 t_2]_{c,d} \cdot \cdots \mid [\varepsilon]_{c,d}) = P_i([t_2]_{c,d} \cdot \cdots \mid [t_1]_{c,d}) \cdot P_i([t_1]_{c,d} \cdots \mid [\varepsilon]_{c,d}) \)

**Proof.** (1) We have \( LS_{c,d}(t \cdot t') = LS_{c,d}(t) \cdot LS_{c,d}(t') \) from Definition 2 for all \( t, t' \) since \( LS_{c,d} \) is a composition of \( map \) and \( filter \). With the definition of \( \sim_{c,d} \), this implies \( [t_1]_{c,d}[t_2]_{c,d} = [t_1 t_2]_{c,d} \).
(2) \( P_i([\varepsilon]_{c,d} \mid [t]_{c,d}) \cdot P_i([t]_{c,d} \cdot \cdots \mid [\varepsilon]_{c,d}) = P_i([\varepsilon]_{c,d} [t]_{c,d}) / P_i([t]_{c,d} \cdot \cdots) / P_i([\varepsilon]_{c,d} \cdot \cdots) \)
\[
= P_i([\varepsilon]_{c,d} [t]_{c,d}) / P_i([\varepsilon]_{c,d} \cdot \cdots) = P_i([t]_{c,d}),
\]
using equation 1 and the fact that all traces begin with \( \varepsilon \in [\varepsilon]_{c,d} \), so \( P_i([\varepsilon]_{c,d} \cdot \cdots) = 1 \).
(3) \( P_i([t_2]_{c,d} \cdot \cdots \mid [t_1]_{c,d}) \cdot P_i([t_1]_{c,d} \cdot \cdots \mid [\varepsilon]_{c,d}) = P_i([t_1 t_2]_{c,d} \cdot \cdots) / P_i([t_1]_{c,d} \cdot \cdots) / P_i([\varepsilon]_{c,d}) \)
\[
= P_i([t_1 t_2]_{c,d} \cdot \cdots) / P_i([\varepsilon]_{c,d}) = P_i([t_1]_{c,d} [t_2]_{c,d} \cdot \cdots \mid [\varepsilon]_{c,d}) = P_i([t_1]_{c,d} [t_2]_{c,d} \cdot \cdots \mid [\varepsilon]_{c,d})
\]
\( \square \)

For the following theorems, let \( op(c) \) denote the operation of an event \( c \), i.e. for \( c = (s, o, s') \), \( op(c) = o \). 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 15 hold. Let \( t_1 \cdot c \cdot t_2 \in T(i) \) and \( t_1' \cdot c' \cdot t_2' \in T(i') \) with \( op(c) = op(c') \), \( cl(op(c)) = L \), \( t_1 \sim_{c,d} t_1' \) and \( i \sim_{c,d} i' \).

Then \( E_{c,d}(c) = E_{c,d}(c') \).

**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 the low part of the trace. 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 the low-observable part of a trace, and since \( t_1 \sim_c t'_1 \), 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 \( E_{cl}(c) = E_{cl}(c') \). \( \square \)

**Lemma 5.** Let \( \mathcal{P} \) be a program that always terminates and let classification rule (a) according to Definition 15 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 \( t'_1 \sim_c t_1 \) 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 \( t'_1 \sim_c t_1 \), \( p \) 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. \( \square \)

We now show that if a program’s PDG can be classified correctly, then PN in the traditional attacker model holds for that program.

**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 11.

Now let \( i, i' \) be two inputs with \( i \sim_c i' \) according to Definition 11, 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 \( t = c_1 \cdot c_2 \cdot \ldots \cdot c_n \) 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

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

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

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

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 \( t' \sim_c t \), making both probabilities 0.

Thus, there is a \( j \) that the factor

\[
P_i(\{c_j\}_{cl} \mid \{c_1 \cdot \ldots \cdot c_{j-1}\}_{cl})
\]
is different for $i$ and $i'$. Let $t_1 = c_1 \cdots c_{j-1}$ and $c = c_j$. Then we have

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

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 low-equivalent to $t_1$.

Without loss of generality we assume

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

(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 low-observable part cannot be empty. The conditional probabilities for the different low-observable parts of low events given $[t_1]_{cl}$ must add up to 1 for $i$ and $i'$, but we have

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

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

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

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 $s := \text{stmt}(o)$ and $s' := \text{stmt}(o')$ an MHP-pair. Let $d := \text{cd}(s, s')$. Then by classification rule (b) with $n = s, m = s'$ all nodes in $\text{path}(d, s)$ are classified as $L$. $o$ is ready to be scheduled for input $i$, so the CFG predecessors of $s$ that were already executed allow $o$ to be executed next. Those are in $\text{path}(d, s)$ since otherwise we would have $d = s$, which is impossible because with MHP($s, s'$) we have MHP($d, s'$), a contradiction to Definition 13. Thus, they are 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 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 could be scheduled. This contradicts the assumption of a probabilistic scheduler. \[\square\]
We now can prove that the conditions of the previous theorem guarantee PN with regards to Giffhorn’s model where only sinks and sources annotated as low are low-observable.

**Corollary 1.** Let $\mathcal{P}$ be a terminating program and $\text{ucl}$ a user annotation for $\mathcal{P}$. Let $\text{cl}$ be a correct classification of its PDG according to Definition 15.

Now let $i \sim_L i'$ according to Definition 11, 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 $\text{ucl}(n) = L$ also fulfill $\text{cl}(n) = L$. Thus, for all traces $t_1, t_2$ we have $t_1 \sim_{cl} t_2 \Rightarrow t_1 \sim_L t_2$. 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,

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

We have thus shown that classification as in definition 15 guarantees PN, and also guarantees that conditions 1*'*, 2*'*, 3*'* hold. But what does this imply about the soundness of 1*'*, 2*'*, 3*'*, and how do these two criteria differ? Of course, a fixpoint iteration according to definition 15 must be implemented anyway, thus it is clear that 1*'*, 2*'*, 3*'* is more expensive. Concerning precision, we found examples where 1*'*, 2*'*, 3*'* is slightly more precise, but these seem to be rare. Concerning soundness, we have the proof for 1*'*, 2*'*, 3*'* from [2] for the case of just one low-nondeterminism, but the general proof only for definition 15.

We believe that definition 15 and the general soundness proof can be improved to be identical in power to 1*'*, 2*'*, 3*'*. For the time being, JOANA offers both criteria (which are both subsumed under the label “RLSOD”) as options.

### 6. 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 a dummy implementation (called an “ideal variant”). It is then checked by IFC that no explicit or implicit flow exists between plain text, secret key and encrypted message; that is, probabilistic noninterference holds for the e-voting system with dummy 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 7; Figure 8 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). Both are marked for JOANA with a @Source annotation. By Definition 15, (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. 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 MHP/(57, 57) ∧ cl(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 \land cl(39) = H. Thus criterion 3* is violated: Giffhorn’s criterion (as well as classical LSOD) thinks there is a probabilistic leak.
public class Setup {
    public static void setup(@Source boolean secret_bit) {
        // HIGH input
        // Public-key encryption for Server
        Decryptor serverDec = new Decryptor();
        Encryptor serverEnc = serverDec.getEncryptor();
        // Creating the server
        Server server = new Server(serverDec, PORT);
        new Thread(server).start();
        // The adversary decides how many clients we create
        while (Environment.untrustedInput() != 0) {
            // determine the value the client encrypts:
            // the adversary gives two values
            byte[] msg1 = Environment.untrustedInputMessage();
            byte[] msg2 = Environment.untrustedInputMessage();
            if (msg1.length != msg2.length) { break; }
            byte[] msg = new byte[msg1.length];
            for(int i = 0; i < msg1.length; ++i)
                msg[i] = (secret_bit ? msg1[i] : msg2[i]);
            // spawn new client thread
            Client client = new Client(serverEnc, msg, HOST, PORT);
            new Thread(client).start();
        }
    }
}

public class KeyPair {
    public byte[] publicKey;
    @Source
    public byte[] privateKey; // HIGH value
}

public final class Decryptor {
    private byte[] privKey;
    private byte[] publKey;
    private MessagePairList log = new MessagePairList();
    public Decryptor() {
        // initialize public and secret (HIGH) keys
        KeyPair keypair = CryptoLib.pke_generateKeyPair();
        publKey = copyOf(keypair.publicKey);
        privKey = copyOf(keypair.privateKey);
    }
}

public class Client implements Runnable {
    private byte[] msg; private Encryptor enc;
    private String hostname; private int port;
    ...
    @Override
    public void run() {
        // encrypt
        byte[] msg_enc = enc.encrypt(msg);
        // send
        long socketID = Network.openConnection(hostname, port);
        Network.sendMessage(socketID, msg_enc);
        Network.closeConnection(socketID);
    }
}

public class Network {
    @Sink // LOW output
    public static void sendMessage(long socketID, byte[] msg) throws NetworkError {
        ...
    }
}

Fig. 8. Relevant parts of the multithreaded encrypted message passing system with security annotations for JOANA.
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: \[ 11 = \text{cda}(57, 57) \].\(^7\) Now it turns out that the initialisation of private keys lies \textit{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!

The second potential probabilistic leak comes from the potential high influence by \texttt{secret\_bit} in line 19 to the low-nondeterministic message sends in line 63. Technically, we have the PDG High chain \(2 \rightarrow 19 \rightarrow 21 \rightarrow 53 \rightarrow 57\), but 57 is manually classified Low. However this second leak candidate is not eliminated by RLSOD, and indeed is a probabilistic leak: since the \texttt{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 \textit{behind} the common dynamic ancestor, but before the low-nondeterminism:

\[ 11 = \text{cda}(57, 57) \rightarrow \text{cfg} 19 \rightarrow \text{cfg} 57. \]

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 \texttt{encrypt} run time does \textit{not} depend on \texttt{msg}, the leak may be ignored.

\(^7\)Note that we indeed need \texttt{cda} instead of \texttt{cdom} here, such that the static \texttt{cda} lies before \textit{all} dynamically possible spawns. JOANA handles this case correctly, as well as handling interprocedural, context-sensitive dynamic ancestors.
JOANA detects both potential probabilistic leaks in about 5 seconds on a standard PC. A JOANA screenshot showing the analysis of the e-voting source code is given in Figure 9; 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. Needless to say, the false alarm is also discovered by the classification criterion (definition 15). 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.

7. 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 immediately guarantees probabilistic non-interference for any scheduler, it is much too strict for multi-threaded programs. In our current work, we considerably improved 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 \( c\text{cda}(n,m) \) 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 [30]. Again, a probabilistic scheduler is assumed. Scheduler-independent approaches can be found in, e.g., [31, 32]. 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 6 left are not probabilistically secure under a round-robin scheduler, their possibilistic notion of bisimilarity require “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.
8. 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


