On Improvements of Low-Deterministic Security

J. Breitner, J. Graf, M. Hecker, M. Mohr, and G. Snelting

Karlsruhe Institute of Technology

Abstract. Low-security observable determinism (LSOD), as introduced by Roscoe and Zdancewic [18,24], is the simplest criterion which guarantees probabilistic noninterference for concurrent programs. But LSOD prohibits any, even secure low-nondeterminism. Giffhorn developed an improvement, named RLSOD, which allows some secure low-nondeterminism, and can handle full Java with high precision [5].
In this paper, we describe a new generalization of RLSOD. By applying aggressive program analysis, in particular dominators for multi-threaded programs, precision can be boosted and false alarms minimized. We explain details of the new algorithm, and provide a soundness proof. The improved RLSOD is integrated into the JOANA tool; a case study is described. We thus demonstrate that low-deterministic security is a highly precise and practically mature software security analysis method.

Keywords: Information Flow Control, Probabilistic Noninterference, Program Analysis

1 Introduction

Information flow control (IFC) analyses a program's source or byte code for leaks, in particular violations of confidentiality and integrity. IFC algorithms usually check some form of noninterference; sound IFC algorithms guarantee to find all possible leaks. For multi-threaded programs, probabilistic noninterference (PN) as introduced in [21,20,19] is the established security criterion. Many algorithms and definitional variations for PN have been proposed, which vary in soundness, precision, scalability, language restrictions, and other features.

One of the oldest and simplest criteria which enforces PN is low-security observational determinism (LSOD), as introduced by Roscoe [18], and improved by Zdancewic, Huisman, and others [24,10]. 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 subtle problems in earlier LSOD algorithms, so Huisman concluded that scheduler-independent PN is not feasible [9]. Worse, LSOD strictly prohibits any, even secure low-nondeterminism – which kills LSOD from a practical viewpoint.

It is the aim of this paper to demonstrate that improvements to LSOD can be devised, which invalidate these earlier objections. An important step was already provided by Giffhorn [5,4] who discovered that

1. an improved definition of low-equivalent traces solves earlier soundness problems for infinite traces and nonterminating programs,

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

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

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

Fig. 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.

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 precise points-to analysis, may-happen-in-parallel 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 RLSOD (Relaxed LSOD) algorithm requires – like many other algorithms, e.g. [21,20] – that the scheduler is probabilistic. RLSOD is integrated into the JOANA IFC tool (`joana.ipd.kit.edu`), which has successfully been applied in various projects [7,5,14,11,12,6].

In this paper, we describe new improvements for RLSOD, which boost precision and reduce false alarms compared to original LSOD and RLSOD. We first recapitulate technical properties of PN, LSOD, and RLSOD. We then introduce the improved criterion, which is based on the notion of dominance in threaded control flow graphs. We explain the definition using examples, provide soundness arguments, and present a case study, namely a prototypical e-voting system with multiple threads. Our work builds heavily on our earlier contributions [7,5], but the current paper is aimed to be self-contained.

2 Probabilistic Noninterference

IFC aims to guarantee that no violations of confidentiality or integrity may occur. For confidentiality, usually all values in input, output, or program states are classified as "high" (secret) or "low" (public), and it is assumed that an attacker can read all low values, but cannot see any high value.[1]

[1] A more detailed discussion of IFC attacker models can be found in e.g. [5]. Note that JOANA allows arbitrary lattices of security classifications, not just the simple

Fig. 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 high output is influenced by low values; in Fig. 1 right, `H` is never copied to `L`, but if the value of `H` is large, probability is higher that "`POST`" is printed instead of "`STPO`".

2.1 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 below.

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: $cl(v) = L \implies s(v) = s'(v)$. 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 [1], 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 $t = (\overline{s}_1, o_1, \underline{s}_1), (\overline{s}_2, o_2, \underline{s}_2), \ldots, (\overline{s}_\nu, o_\nu, \underline{s}_\nu), \ldots$, where the o_ν are operations (i.e. dynamically executed program statements c_ν; we write $stmt(o_\nu) = c_\nu$). $\overline{s}_\nu, \underline{s}_\nu$ are the states before resp. after executing o_ν. For the time being we assume traces to be terminating; subtleties of nontermination are discussed later.

For PN, the notion of low-equivalent traces is essential. Classically, traces are low equivalent if for every $(\overline{s}_\nu, o_\nu, \underline{s}_\nu) \in t$, $(\overline{s'_\nu}, o_\nu, \underline{s'_\nu}) \in t'$, it holds that $\overline{s}_\nu \sim_L \overline{s'_\nu}$ and $\underline{s}_\nu, \sim_L \underline{s'_\nu}$. This definition enforces a rather restrictive lock-step execution of both traces. Later definitions (e.g. [20]) use stutter equivalence instead of lock-step equivalence; thus allowing one execution to run faster than the other ("stuttering"

$\bot = L \leq H = \top$ lattice. Note also that integrity is dual to confidentiality, but will not be discussed here. JOANA can handle both.

```
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. 2. Left: insecure program, obvious explicit leak. Middle: secure program, RLSOD + flow sensitivity avoid false alarm. Right: only iRLSOD avoids false alarm.

means that one trace performs additional operations which do not affect public behaviour). In our flow-sensitive setting, we achieve the same effect by demanding that not only program variables are classified, but also all program statements ($cl(c) = H$ or $cl(c) = L$), and thus operations in traces: $cl(o) = cl(stmt(o))$. Note that it is not necessary for the engineer to provide classifications for all program statements, as most of the $cl(c)$ can be computed automatically (see below). Low equivalence then includes filtering out high operations from traces. This leads to

Definition 2. *1. The low-observable part of an event is defined as*

$$E_L((\overline{s}, o, \underline{s})) = \begin{cases} (\overline{s}\mid_{use(o)}, o, \underline{s}\mid_{def(o)}), & \text{if } cl(stmt(o)) = L \\ \epsilon, & \text{otherwise} \end{cases}$$

where $def(o), use(o)$ are the variables defined (i.e. assigned) resp. used in o.

2. The low-observable subtrace of trace t is

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

3. Traces t, t' are low-equivalent, written $t \sim_L t'$, if $E_L(t) = E_L(t')$.

Note 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; resulting in more traces to be low-equivalent without compromising soundness. This subtle observation is another reason why flow-sensitive IFC is more precise.

PN is called "probabilistic", because it essentially depends of the probabilities for certain traces under certain inputs: $P_i(t)$ is the probability that a specific trace t is executed under input i; and $P_i([t]_L)$ is the probability that some trace $t' \in [t]_L$ (i.e. $t' \sim_L t$) is executed under i. Note that the $t' \in [t]_L$ cannot be distinguished by an attacker, as all $t' \in [t]_L$ have the same public behaviour.
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. Inputs i, i' are low equivalent ($i \sim_L i'$) if they coincide on low values: $cl(i_\nu) = L \wedge cl(i'_\nu) = L \implies i_\nu = i'_\nu$. The definition relies on our flow-sensitive $t \sim_L t'$.

Definition 3 (Probabilistic noninterference). *Let i, i' be input streams; let $T(i)$ be the set of all possible traces of program $\mathcal{P}$ for input i, $\Theta = T(i) \cup T(i')$. 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.

As $[t]_L$ is discrete (in fact recursively enumerable), P_i is a discrete probability distribution, hence $P_i([t]_L) = \sum_{t' \in [t]_L} P_i(t')$. Thus the PN condition can be rewritten 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 $t \in \Theta$ there are only two possibilities: $\ldots$`print("PO")`$\ldots$`print("ST")`$\ldots \in t$ or $\ldots$`print("ST")`$\ldots$`print("PO")`$\ldots \in t$. There are no other low events or low output, hence there are only two equivalence classes $[t]^1_L = \{t' \mid \ldots \texttt{print}(\text{"}\texttt{PO}\text{"})\ldots\texttt{print}(\text{"}\texttt{ST}\text{"})\ldots \in t'\}$ and $[t]^2_L = \{t' \mid \ldots \texttt{print}(\text{"}\texttt{ST}\text{"})\ldots\texttt{print}(\text{"}\texttt{PO}\text{"})\ldots \in t'\}$. 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 explicitely determine the P_i, where the Markow chain models the probabilistic state transitions of a program, perhaps together with a specific scheduler [20,15]. Worse, the sums might be infinite (but will always converge). Practical examples with explicit probabilities can be found in [5,4]. Here, 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 we may conclude $i \sim_L i'$ and apply the PN definition. Let $t'' \in \Theta$. As $\mathcal{P}$ is sequential, $t'' = t \in_1 T(i)$ or $t'' = t' \in_1 T(i')$. WloG let $t'' = t$. Due to PN, $P_i([t]_L) = P_{i'}([t]_L)$, due to sequentiality $P_i([t]_L) = P_i(t) = P_{i'}(t') = 1$, thus $P_{i'}([t]_L) = P_{i'}(t') = 1$. 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 $[\![\mathcal{P}]\!](s) \sim_L [\![\mathcal{P}]\!](s')$. □

2.3 Low-deterministic Security

LSOD is the oldest and still the simplest criterion which enforces PN. LSOD demands that low-equivalent inputs produce low-equivalent traces. LSOD is

scheduler independent and implies PN (see lemma 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 4 (Low-security observational determinism). *Let i, i' be input streams, Θ as above. LSOD holds iff*

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

Under LSOD, all traces t for input i are low-equivalent: $T(i) \subseteq [t]_L$, because $\forall t' \in T(i) \colon t' \sim_L t$. 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 $P_i([t]_L) = P_{i'}([t]_L)$ holds. □

Zdancewic [24] 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 explicitely 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 `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 RLSOD

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

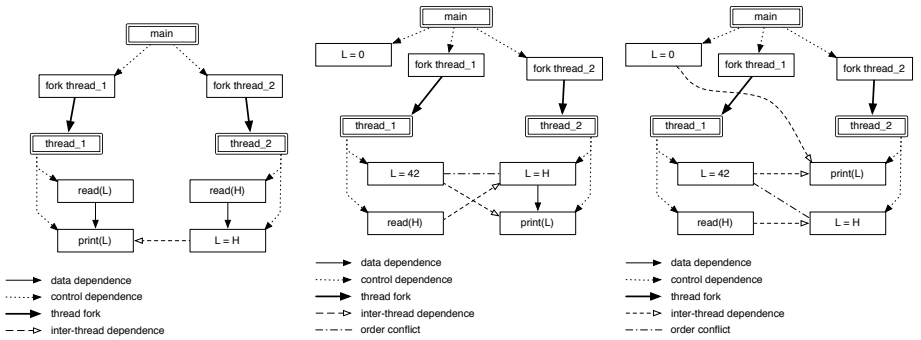

Fig. 3. Left to right: PDGs for Figure 1 middle, and for Figure 2 left and middle.

3.1 PDGs for IFC

Snelting et al. introduced Program Dependence Graphs (PDGs) as a device to check integrity of software [22]. Later the approach was expanded into the JOANA IFC project. It was shown that PDGs guarantee sequential noninterference [23], and that PDGs provide improved precision as they are naturally flow- and context-sensitive [7].

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 [7]. We will not discuss PDG details; it is sufficient to know the *Slicing Theorem*:

Theorem [8]. 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.

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.

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 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 `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 [23]. 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.

Note that the slicing theorem does not cover physical side channels such as power consumption profiles, nor does it cover corrupt schedulers or defective hardware; it only covers "genuine" program behaviour.

In the following, we will need some definitions related to PDGs. For more details on PDGs, MHP, flow- context-, object- and time-sensitivity, see [7].

Definition 5. *1. Let $G = (N, \to)$ be a PDG, where N consists of program statements and expressions, and $\to$ comprises data dependencies, control dependencies, summary dependencies, and inter-thread dependencies. The (context-sensitive) backward slice for $n \in N$ is defined as*

$$BS(n) = \{m \mid m \to^*_{realizeable} n\}$$

where $\to^*_{realizeable}$ includes only context- object- and (optionally) time-sensitive paths in the PDG [7].

2. All input and output statements $n \in N$ are assumed to be classified as $cl(n) = H$ or $cl(n) = L$. Other PDG nodes need not be explicitely classified, but a classification can be computed via the flow equation

$$cl(n) = \bigsqcup_{m \to n} cl(m).$$

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

3. We write $\mathrm{MHP}(n, m)$ if MHP analysis concludes that n and m may be executed in parallel. Thus by interleaving there may be traces t, t' where $t = \ldots (\overline{s_n}, o_n, \underline{s_n}) \ldots (\overline{s_m}, o_m, \underline{s_m}) \ldots$, $t' = \ldots (\overline{s_m}, o_m, \underline{s_m}) \ldots (\overline{s_n}, o_n, \underline{s_n}) \ldots$.

Concerning cl it is important to note that PDGs are automatically flow-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 $cl(n)$ together with the global flow constraints $cl(n) = \bigsqcup_{m \to n} cl(m)$. The latter can easily be computed or checked by a fixpoint iteration on the PDG [7].

3.2 Relaxed LSOD

In his 2012 thesis, Giffhorn applied PDGs to PN. He showed that PDGs can naturally be used to check the LSOD property, and provided a soundness proof as well as an implementation for JOANA [4]. Giffhorn also found the first optimization relaxing LSOD's strict low-determinism, named RLSOD.

One issue was to plug soundness leaks which had been found in some earlier approaches to LSOD. In particular, treatment of nontermination had proven to be tricky. Giffhorn provided a new definition for low-equivalent traces, where $t \sim_L t'$ 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 [5].

It turned out that the last condition not only avoids previous soundness leaks, but can precisely be characterized by dynamic control dependencies in traces [5]. 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. Formally Giffhorn's LSOD criterion reads as follows:

Theorem 1. *Let $n, n', n'' \in N$ be PDG nodes. LSOD holds if*

1. $\forall n, n' \colon\ cl(n) = L \wedge cl(n') = H \implies n' \notin BS(n)$,
2. $\forall n, n', n'' \colon\ \mathrm{MHP}(n, n') \wedge \exists v \in def(n) \cap (def(n') \cup use(n')) \wedge cl(n'') = L$
   $\implies n \notin BS(n'') \wedge n' \notin BS(n'')$,
3. $\forall n, n' \colon\ \mathrm{MHP}(n, n') \implies cl(n) = H \vee cl(n') = H$.

Proof. For proof and implementation details, see [5]. □

Applying this 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 according to condition 3: we have MHP(`L = 42;`, `print(L);`), which causes a false alarm.

The example motivates the RLSOD 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 $cl(n'') = H$, to n or n', where $cl(n) = cl(n') = L$ and $\mathrm{MHP}(n, n')$. 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 the RLSOD criterion, which replaces condition 3 above by

3'. $\forall n, n' \colon\ \mathrm{MHP}(n, n') \wedge \exists n'' \colon cl(n'') = H \wedge (n'' \rightarrow^*_{CFG} n \vee n'' \rightarrow^*_{CFG} n')$
$\implies cl(n) = H \vee cl(n') = H$.

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

3'. $\forall n, n'\colon \ \mathrm{MHP}(n, n') \wedge cl(n) = L \wedge cl(n') = L$
$\implies \forall n'' \in START \rightarrow^*_{CFG} n \cup START \rightarrow^*_{CFG} n'\colon cl(n'') = L.$

In fact the same argument not only holds for execution order conflicts, but also for data races: 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'. $\forall n, n', n''\colon \ \mathrm{MHP}(n, n') \wedge \exists n'''\colon cl(n''') = H \wedge (n''' \rightarrow^*_{CFG} n \vee n''' \rightarrow^*_{CFG} n')$
$\wedge \exists v \in def(n) \cap (def(n') \cup use(n')) \wedge cl(n'') = L$
$\implies n, n' \notin BS(n'').$

By contraposition, we obtain the more practical form

2'. $\forall n, n', n''\colon \ \mathrm{MHP}(n, n') \wedge \exists v \in def(n) \cap (def(n') \cup use(n'))$
$\wedge cl(n'') = L \wedge \big(n \in BS(n'') \vee n' \in BS(n'')\big)$
$\implies \forall n''' \in START \rightarrow^*_{CFG} n \cup START \rightarrow^*_{CFG} n'\colon cl(n''') = L.$

In fact RLSOD, as currently implemented in JOANA, uses even more precise refinements of conditions 2' and 3' (see [4], pp. 200ff), which we however omit due to lack of space. Figure 1 right is not RLSOD, 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 RLSOD, because the low-nondeterminism in line 6 resp. 9 can not 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.[2] Indeed the program is PN. Figure 2 right is however not RLSOD, because the initial `read(H2)` will reach any other statement. But the program is PN, because `H2` does not influence any later low statement! The example shows that RLSOD does indeed reduce false alarms, but it effectively 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 RLSOD 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 for RLSOD, as RLSOD is a special case of the improvement which will be discussed in the next section.

4 Improving RLSOD

In the following, we will generalize condition 3' to obtain a much more precise "iRLSOD" criterion. The same improvement can be applied to condition 2' as

[2] That is, 2' as in [4] holds; the slightly less precise, but simpler 2' condition in the current paper is violated, but 3' as defined in the current paper holds. We thank C. Hammer and his students for this subtle observation.

well – the usage of dominators (see below) and the soundness proof are essentially "isomorphic". It is only for reasons of space and readability that in this paper we only describe the improvement of 3'. For the same reasons, we stick to definitions 2' and 3', even though JOANA uses a slightly more precise variant (see above); the iRLSOD improvement works the same with the more precise 2' and 3'.

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 $[t]^1_L = \{t' \mid \ldots t' = \texttt{print("PO")} \ldots \texttt{print("ST")} \ldots\}$ and $[t]^2_L = \{t' \mid t' = \ldots \texttt{print("ST")} \ldots \texttt{print("PO")} \ldots\}$. Likewise, $\mathcal{P}_2$ has two low classes $[t]^1_L = \{t' \mid t' = \ldots \texttt{L} = \texttt{42} \ldots \texttt{print(42)} \ldots\}$ and $[t]^2_L = \{t' \mid t' = \ldots \texttt{print(0)} \ldots \texttt{L} = \texttt{42} \ldots\}$. 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 $\mathcal{P}_2$, $P_i([t]^{1,2}_L) = P_{i'}([t]^{1,2}_L)$ holds!

Technically, $\mathcal{P}_2$ contains a point c which *dominates* both low-nondeterministic statements $n \equiv \texttt{L} = \texttt{42};$, $m \equiv \texttt{print(L)}$, 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 $n \equiv \texttt{print("PO")}$ there is no high event, while on the path to $m \equiv \texttt{print("ST")}$ 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.

4.1 Improving Condition 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 5). We will now provide precise definitions for this idea.

Definition 6 (Common dominator). *Let two statements $n, m \in \mathcal{P}$ be given.*

1. *Statement c is a dominator for n, written c dom n, if c occurs on every CFG path from $START$ to n.*

2. *Statement c is a common dominator for n, m, written c cdom (n, m), if c dom $n \wedge c$ dom m.*
3. *If c cdom (n, m) and $\forall c'$ cdom (n, m): c' dom c, then c is called an immediate common dominator.*

Efficient algorithms for computing dominators can be found in many compiler textbooks. Intraprocedural immediate dominators are unique and give rise to the dominator tree; for unique immediate common dominators we write $c = idom(n, m)$.[3] Note that $START$ itself is a (trivial) common dominator for every n, m. iRLSOD works with any common dominator. We thus assume a function *cdom* which for every statement pair returns a common dominator, and write $c = cdom(n, m)$. Note that the implementation of *cdom* may depend on the precision requirements, but once a specific *cdom* is chosen, c depends solely on n and m. We are now ready to formally define the improved RLSOD criterion.

Definition 7 (iRLSOD). *iRLSOD holds if LSOD conditions 1 and 2 hold for all PDG nodes, and if*

3". $\forall n, n'\colon \ \mathrm{MHP}(n, n') \wedge cl(n) = cl(n') = L \wedge c = cdom(n, n')$
$\implies \forall n'' \in c \to^*_{CFG} n \cup c \to^*_{CFG} n'\colon cl(n'') = L.$

iRLSOD is most precise (generates the least false alarms) if $cdom = idom$, because in this case it demands $cl(n'') = L$ for the smallest set of nodes "behind" the common dominator. Figure 4 illustrates the iRLSOD definition. Note that the original RLSOD trivially fulfils condition 3", where *cdom* always returns $START$. Thus iRLSOD is a true generalization.

4.2 Classification Revisited

Consider the program in Figure 5 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 is not RLSOD, as the print statements can be reached from the high statement in line 3 (middle). Applying iRLSOD, the common dominator for the two print statements is line 10.

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

But according to the flow equation, only line 3 is explicitely high and only lines 4, 7, 8 are PDG-reachable from 3. Thus $cl(10) = L$. Hence iRLSOD would

[3] In programs with procedures and threads, immediate dominators may not be unique due to context-sensitivity [2]. Likewise, the dominator definition must be extended if the same thread can be spawned several times. Both issues are not discussed here.

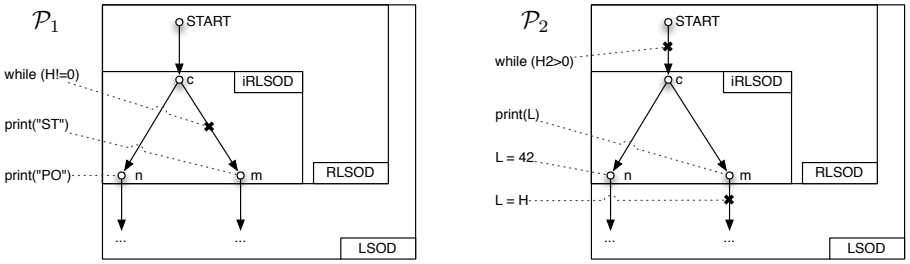

Fig. 4. Visualization of LSOD vs. RLSOD vs. iRLSOD. CFGs for Figure 1 right resp. Figure 2 right are sketched. n/m produces low nondeterminism, c is the common dominator. LSOD prohibits any low nondeterminism; RLSOD allows low nondeterminism which is not reachable by any high events; iRLSOD allows low 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 iRLSOD is much more precise.

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 standard flow equation – it must be extended for low nondeterminism.

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 $idom(n, m) = c$, and the path $c \to^*_{CFG} n$ violates iRLSOD – 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.[4]

Definition 8 (Classification in PDGs). *A PDG $G = (N, \to)$ is classified correctly, if*

1. $\forall n \in N\colon cl(n) \geq \bigsqcup_{m \to n} cl(m)$,
2. $\forall n, m \in N\colon \mathrm{MHP}(n, m) \wedge c = idom(n, m) \wedge \exists c' \in c \to^*_{CFG} n, cl(c') = H \implies cl(n) = H$.

In condition 1, $\geq$ must be used because 2 can push $cl(n)$ higher than $\bigsqcup_{m \to n} cl(m)$. In the example, the rule enforces line 10 to be classified high, as we have $\mathrm{MHP}(10, 18) = 6$, and on the path from 6 to 10, lines 7 and 8 are high.

4.3 Soundness Arguments

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

[4] In case n was manually classified low, a trivial explicit leak has been discovered. Same for the standard flow equation [7].

```
1  thread1(){
2    if (H) {
3      skip;};
4    fork Thread2();
5    print(17);
6  }
7
8  thread2() {
9    print(42);
10 }
```

```
1  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 }
```

```
17 thread2() {
18   tmp2 = 100;
19 }
20
21 thread3() {
22   print(L2);
23 }
```

Fig. 5. Left: deterministic round-robin scheduling may leak. Middle/Right: a leak which goes undiscovered if classification of statements is incomplete.

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. [20]. Indeed a malicious scheduler can read high values to construct an explicit flow by scheduling, as in `{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 5 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→5 resp. 5→9, leaking `H`. However under the assumption of truly probabilistic scheduling, Figure 5 left is iRLSOD.

In the following, let $t_1 \cdots$ be the set of traces beginning with 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. We denote with $P_i(t_2 \mid t_1) = P_i(t_1 \cdot t_2)/P_i(t_1 \cdots)$ the conditional probability that after t_1, execution continues with t_2. This notion extends to sets of traces: $P_i(T' \mid T) = \sum_{t \in T' \cdot T} P_i(t) / \sum_{t \in T} P_i(t \cdots)$.

For the following soundness theorem, we assume that there is only one point of low-nondeterminism. In this case LSOD conditions 1, 2 and 3 hold for the whole program, except for the one point where low-nondeterminism is possible and only the iRLSOD condition 3" holds.

Theorem 2. *Let iRLSOD hold for $\mathcal{P}$, where $\mathcal{P}$ contains only one pair n, n' of low-nondeterministic statements:* $\mathrm{MHP}(n, n'), cl(n) = cl(n') = L$.

Now let $i \sim_L i'$, let $t \in \Theta$. Then

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

Proof (sketch). If t contains neither n nor n', LSOD holds and thus the PN condition $P_i([t]_L) = P_{i'}([t]_L)$ trivially holds.

Thus we assume, without loss of generality, that n occurs on t, and before a possible occurence of n'. Let $c = cdom(n, n')$.

The iRLSOD conditions ensures that t can be decomposed as $t_1 \cdot c \cdot t_2 \cdot n \cdot t_3$, and furthermore all low events on t_2 are on the control path from c to n or n', while any high events are from possible other threads. These other threads cannot have any low operations in t_2, as that would form another MHP-pair with n and n'. Correspondingly, $i = i_1 i_c i_2 i_n i_3$, where i_ν is consumed by t_ν.

Any trace $t' \sim_L t$ necessarily contains c and n and can be decomposed analogously and uniquely, with $t'_1 \sim_L t_1$, $t'_2 \sim_L t_2$ and $t'_3 \sim_L t_3$. Therefore, we have

$$P_i([t]_L) = P_i([t_1 \cdots]_L) \cdot P_i(c \cdot [t_2]_L \cdot n \cdots \mid [t_1]_L) \cdot P_i(t_3 \mid [t_1 \cdot c \cdot t_2]_L \cdot n).$$

by the chain rule for conditional probabilities, and the same for $i' = i'_1 i'_c i'_2 i'_n i'_3$.

We show $P_i([t]_L) = P_{i'}([t]_L)$ by equating these factors:

- We have $P_i([t_1 \cdots]_L) = P_{i'}([t_1 \cdots]_L)$: There is no low nondeterminism in the part of the CFG that produced this initial segment of the trace, and by the usual soundness argument for LSOD (cmp. Lemma 2), we find that $P_i([t_1 \cdots]_L) = 1$, and analogously for i'.
- We have $P_i(c \cdot [t_2]_L \cdot n \cdots \mid [t_1]_L) = P_{i'}(c \cdot [t_2]_L \cdot n \cdots \mid [t_1]_L)$: If there were no other, high threads, $c \cdot t_2 \cdot n$ would consist exclusively of low events. Since we assume a scheduler that does neither maintain its own state nor looks at the value of variables, the probabilities depend only on the part of i resp. i' that is consumed by the trace between c and n, namely i_2 resp. i'_2. As t_2, t'_2 contain only low operations, i_2, i'_2 is also classified low; and as we have $i \sim_L i'$, $i_2 = i'_2$ must hold. Therefore the probabilities are equal.
  If there are other threads, which necessarily only execute high events in this part of the execution, then these may slow down t_2 resp. t'_2 (similar to "stuttering"), but, as we assume a fair scheduler, do not change their relative probabilities. Therefore, these differences are factored out by considering low equivalency classes and equality holds in this case as well.
- For $P_i(t_3 \mid [t_1 \cdot c \cdot t_2]_L \cdot n) = P_i(t_3 \mid [t_1 \cdot c \cdot t_2]_L \cdot n)$ we are again in the situation of no low nondeterminism, as any possible nondeterminism is caused by the MHP-pair (n, n'), so analogously to the initial segment, both probabilities are one.

□

Note that the restriction to a single low-nondeterministic pair still covers many applications. An inductive proof for the general case (more than one low nondeterminism pair) is work in progress.

Corollary 1. *RLSOD is sound.*

Proof. RLSOD is a special case of iRLSOD: choose $cdom(n, n') = START$. □

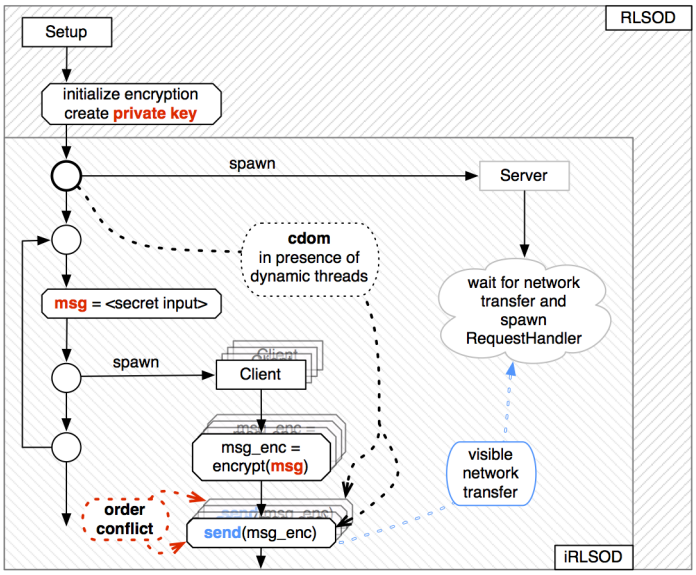

Fig. 6. CFG structure of the multithreaded server-client based message transfer.

5 Case Study: E-Voting

In the following, we will apply RLSOD/iRLSOD 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 proof 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 [12,11].

The example uses a multithreaded client-server architecture to send encrypted messages over the network. It consists of 550LoC with 16 classes. The interprocedural control flow is sketched in Figure 6; Figure 7 contains relevant parts of the code. The main thread starts in class `Setup` in line 3ff: 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 3) that decides about the content of the message;

and (2) the private key of the encryption (line 33). Both are marked for JOANA with a `@Source` annotation. By Definition 8, (2) propagates to lines 44, 46, 5, 8 and 9 which are also classified High. Likewise, (1) propagates to lines 21 and 24, wich are thus High as well.

As information sent over network is visible to the attacker, calls to the method `sendMessage` (line 66f) are marked as a LOW `@Sink`. JOANA was started in RLSOD 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 in RLSOD mode; one is later uncovered by iRLSOD 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, as 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 44, hence RLSOD will cause an alarm (cmp. RLSOD criterion 3). Technically, we have $\mathrm{MHP}(66, 66) \wedge cl(66) = L$; that is, line 66 is low-nondeterministic with itself (because the same thread is spawned several times). Furthermore, $START \rightarrow^*_{CFG} 44 \rightarrow^*_{CFG} 66 \wedge cl(44) = H$. Thus RLSOD criterion 3' is violated.

Now let us apply iRLSOD to this leak. The dominator for the low-nondeterministic message sends in line 66 is located at the loop header: $12 = cdom(66, 66)$.[5] Now it turns out that the initialisation of private keys lies *before* this common dominator: lines 33, 44, 46, 5, 8, and 9 context-sensitively dominate line 12. Thus by iRLSOD 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 `secret_bit` in line 21 to the low-nondeterministic message sends in line 73. Technically, we have the PDG High chain $3 \rightarrow 21 \rightarrow 24 \rightarrow 62 \rightarrow 66$, but 66 is manually classified Low. However this second leak candidate is not eliminated by iRLSOD, 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, iRLSOD discovers this subtle leak because the high operation which accesses the secret bit lies *behind* the common dominator, but before the low-nondeterminism: $12 = cdom(66, 66) \rightarrow^*_{CFG} 21 \rightarrow^*_{CFG} 66$.

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 leaks in about 5 seconds on a standard PC.

[5] Note that in case of dynamically created threads, the definition of common dominator must be extended, such that the static *cdom* lies before *all* dynamically possible spawns. This extension for dynamic threads is not covered by definition 6, but implemented in JOANA. JOANA also handles interprocedural, context-sensitive dominators.

```
1  public class Setup {
2
3    public static void setup(@Source boolean secret_bit) { // HIGH input
4      // Public-key encryption functionality 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
11     // The adversary decides how many clients we create
12     while (Environment.untrustedInput() != 0) {
13       // determine the value the client encrypts:
14       // the adversary gives two values
15       byte[] msg1 = Environment.untrustedInputMessage();
16       byte[] msg2 = Environment.untrustedInputMessage();
17       if (msg1.length != msg2.length) { break; }
18
19       byte[] msg = new byte[msg1.length];
20       for(int i = 0; i < msg1.length; ++i)
21         msg[i] = (secret_bit ? msg1[i] : msg2[i]);
22
23       // spawn new client thread
24       Client client = new Client(serverEnc, msg, HOST, PORT);
25       new Thread(client).start();
26     }
27   }
28 }
29
30 public class KeyPair {
31   public byte[] publicKey;
32   @Source
33   public byte[] privateKey; // HIGH value
34 }
35
36 public final class Decryptor {
37
38   private byte[] privKey;
39   private byte[] publKey;
40   private MessagePairList log = new MessagePairList();
41
42   public Decryptor() {
43     // initialize public and secret (HIGH) keys
44     KeyPair keypair = CryptoLib.pke_generateKeyPair();
45     publKey = copyOf(keypair.publicKey);
46     privKey = copyOf(keypair.privateKey);
47   }
48
49   ...
50
51 }
52
53 public class Client implements Runnable {
54
55   private byte[] msg; private Encryptor enc;
56   private String hostname; private int port;
57   ...
58
59   @Override
60   public void run() {
61     // encrypt
62     byte[] msg_enc = enc.encrypt(msg);
63
64     // send
65     long socketID = Network.openConnection(hostname, port);
66     Network.sendMessage(socketID, msg_enc);
67     Network.closeConnection(socketID);
68   }
69 }
70
71 public class Network {
72
73   @Sink // LOW output
74   public static void sendMessage(long socketID, byte[] msg) throws NetworkError {
75     ...
76   }
77   ...
78 }
```

Fig. 7. Relevant parts of the multithreaded encrypted message passing system with security annotations for JOANA.

6 Related Work

Zdancewic's work [24] 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 (i)RLSOD to truly probabilistic schedulers). Smith [20] 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. This *weak* probabilistic bisimulation allows the execution time of threads to depend on secret input, as long as it is not made observable by writing to public variables. 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 *cdom* (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 non-interference of concurrent programs based on type systems benefit from various degrees of compositionality, a good study of which is given in [16]. Again, a probabilistic scheduler is assumed. Scheduler-independent approaches can be found in, e.g., [13,17]. 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 5 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 iRLSOD this is more restrictive for programs, but less restrictive on scheduling.

7 Future Work

RLSOD is already part of JOANA; we currently integrate iRLSOD into the system. We will thus be able to provide empirical precision comparisons between iRLSOD, RLSOD, and LSOD. Another issue is a generalization of Theorem 2 for multiple *MHP* pairs with corresponding multiple common dominators.

One issue which might push precision even further is lock sensitivity. The current MHP and dominator algorithms analyse thread invocations in a context-sensitive manner, but do ignore explicit locks. We started an integration of

Müller-Olm's lock-sensitive Dynamic Pushdown Networks [3] into MHP, which sometimes can eliminate inter-thread dependences. The dominator computation for multi-threaded programs could profit from lock-sensitivity as well.

8 Conclusion

JOANA can handle full Java with arbitrary threads, while being sound and scaling to several 10k 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.

Acknowledgements. This work was partially supported by Deutsche Forschungsgemeinschaft in the scope of SPP "Reliably Secure Software Systems", and by BMBF in the scope of the KASTEL project.

References

1. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Proc. ESORICS. LNCS, vol. 5283, pp. 333–348 (2008)
2. De Sutter, B., Van Put, L., De Bosschere, K.: A practical interprocedural dominance algorithm. ACM Trans. Program. Lang. Syst. 29(4) (Aug 2007)
3. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: VMCAI. pp. 199–213 (2011)
4. Giffhorn, D.: Slicing of Concurrent Programs and its Application to Information Flow Control. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (May 2012)
5. Giffhorn, D., Snelting, G.: A new algorithm for low-deterministic security. International Journal of Information Security 14(3), 263–287 (Apr 2015)
6. Graf, J., Hecker, M., Mohr, M., Snelting, G.: Checking applications using security APIs with JOANA (Jul 2015), `http://www.dsi.unive.it/~focardi/ASA8/`, 8th International Workshop on Analysis of Security APIs
7. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8(6), 399–422 (December 2009)
8. Horwitz, S., Prins, J., Reps, T.: On the adequacy of program dependence graphs for representing programs. In: Proc. POPL '88. pp. 146–157. ACM, New York, NY, USA (1988)
9. Huisman, M., Ngo, T.: Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In: Proc. Formal Verification of Object-Oriented Systems (2011)
10. Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational determinism. In: Proc. 19th CSFW. p. 3. IEEE (2006)
11. Küsters, R., Scapin, E., Truderung, T., Graf, J.: Extending and applying a framework for the cryptographic verification of Java programs. In: Proc. POST 2014. pp. 220–239. LNCS 8424, Springer (2014)

12. Küsters, R., Truderung, T., Graf, J.: A framework for the cryptographic verification of Java-like programs. In: Computer Security Foundations Symposium (CSF), 2012 IEEE 25th. IEEE Computer Society (Jun 2012)
13. Mantel, H., Sudbrock, H.: Flexible scheduler-independent security. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) Computer Security – ESORICS 2010, Lecture Notes in Computer Science, vol. 6345, pp. 116–133. Springer Berlin Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15497-3_8
14. Mohr, M., Graf, J., Hecker, M.: JoDroid: Adding Android support to a static information flow control tool. In: Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. Mäarz 2015. CEUR Workshop Proceedings, vol. 1337, pp. 140–145. CEUR-WS.org (2015)
15. Ngo, T.M.: Qualitative and quantitative information flow analysis for multi-threaded programs. Ph.D. thesis, University of Enschede (april 2014)
16. Popescu, A., Hölzl, J., Nipkow, T.: Formalizing probabilistic noninterference. In: Certified Programs and Proofs – Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings. pp. 259–275 (2013)
17. Popescu, A., Hölzl, J., Nipkow, T.: Noninterfering schedulers. In: Heckel, R., Milius, S. (eds.) Algebra and Coalgebra in Computer Science, Lecture Notes in Computer Science, vol. 8089, pp. 236–252. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40206-7_18
18. Roscoe, A.W., Woodcock, J., Wulf, L.: Non-interference through determinism. In: ESORICS. LNCS, vol. 875, pp. 33–53 (1994)
19. Sabelfeld, A., Sands, D.: 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. pp. 200–214 (2000)
20. Smith, G.: Improved typings for probabilistic noninterference in a multi-threaded language. J. Comput. Secur. 14(6), 591–623 (2006), http://iospress.metapress.com/content/4wt8erpe5eqkc0df
21. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proc. POPL '98. pp. 355–364. ACM (January 1998)
22. Snelting, G.: Combining slicing and constraint solving for validation of measurement software. In: SAS '96: Proceedings of the Third International Symposium on Static Analysis. pp. 332–348. Springer-Verlag, London, UK (1996)
23. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Trans. Softw. Eng. Methodol. 15(4), 410–457 (2006)
24. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proc. CSFW. pp. 29–43. IEEE (2003)