Noname manuscript No.
(will be inserted by the editor)

Dennis Giffhorn · Gregor Snelting

A New Algorithm for Low-Deterministic Security

the date of receipt and acceptance should be inserted later

Abstract We present a new algorithm for checking probabilistic noninterference in concurrent programs. The algorithm, named RLSOD, is based on the *Low-Security Observational Determinism* criterion. It utilizes program dependence graphs for concurrent programs, and is flow-sensitive, context-sensitive, object-sensitive, and optionally time-sensitive. Due to a new definition of low-equivalency for infinite traces, the algorithm avoids restrictions or soundness leaks of previous approaches. A soundness proof is provided. Flow-sensitivity turns out to be the key to precision, and avoids prohibition of useful nondeterminism. The algorithm has been implemented for full Java bytecode with unlimited threads. Precision and scalability have been experimentally validated.

Keywords software security, noninterference, program dependence graph, information flow control

1 Introduction

Information flow control discovers software security leaks by analysing the source or machine code. Information flow control for multi-threaded programs is challenging, as it must prevent possibilistic or probabilistic information leaks. Both types of leaks depend on the interleaving of concurrent threads on a processor: possibilistic leaks may or may not occur depending on a specific interleaving, while probabilistic leaks exploit the probability distribution of interleaving orders. Figure 1 presents an example: the bottom left program has a possibilistic leak, e.g., for interleaving order 2, 6, 7, 3, which causes the secret PIN to be printed on public output. The program to the right has no possibilistic channel leaking PIN information, because the printed value of x is always 0 or 1. But the PIN's value may alter the probabilities of these outputs, because the running time of the loop may influence the interleaving order of the two assignments to x. Thus a secret value changes the probability of a public output – a probabilistic leak.

```
1 void main():
2   x = inputPIN();
3   if (x < 1234)
4       print(0);
5   y = x;
6   print(y);
```

```
1 void thread_1():
2   x = input();
3   print(x);
4
5 void thread_2():
6   y = inputPIN();
7   x = y;
```

```
1 void thread_1():
2   x = 0;
3   print(x);
4
5 void thread_2():
6   y = inputPIN();
7   while (y != 0)
8       y--;
9   x = 1;
```

Fig. 1 Examples for explicit and implicit leaks (top left), for a possibilistic leak (bottom left), and for a probabilistic leak (right).

Information flow control (IFC) aims at discovering all such security leaks. Most IFC approaches check some form of noninterference [40], and to this end classify program variables, input and output as high (secret) or low (public). *Probabilistic Noninterference* [44,42,41,43,30] is the established security criterion for concurrent programs. It is difficult to guarantee probabilistic noninterference, as an IFC must in principle check all possible interleavings and their impact on execution probabilities. This is why some analysis algorithms for probabilistic noninterfererence put severe restrictions on program or scheduler behaviour.

One specific form of probabilistic noninterference (PN), however, is scheduler-independent: *Low-Security Observational Determinism* demands that for a program which

This work was partially supported by DFG grants Sn11/9-2 and Sn11/12-1 in the scope of the priority program "Reliably Secure Software Systems". It is based on [8] with additional contributions by the second author. A preliminary version was published as an unreviewed technical report [10].

Dennis Giffhorn
Karlsruhe Institute of Technology, Germany
E-mail: giffhorn@ipd.info.uni-karlsruhe.de

Gregor Snelting
Karlsruhe Institute of Technology, Germany
E-mail: gregor.snelting@kit.edu

runs on two low-equivalent inputs, all possible traces are low-equivalent [39,53,21]. Traces log all operations (events) and memory states for a given run and interleaving; low-equivalent inputs coincide on low input values, and low-equivalent traces coincide on operations using low variables. Thus low-security observational determinism (LSOD) in fact demands that *execution order conflicts between low events are disallowed, if they may be influenced by high events.* The following criterion is sufficient to guarantee LSOD [53]: 1. program parts contributing to low-observable behaviour are conflict-free, that is, the program is low-observable deterministic; 2. implicit or explicit flows do not leak high data to low-observable behaviour. Earlier research [53,21] has shown that the LSOD criterion guarantees PN. But several attempts to devise program analysis algorithms for LSOD turned out to be unsound, unprecise, or very restrictive. In particular, simplistic LSOD will absolutely prohibit any (even secure) low-nondeterminism. Hence LSOD never gained popularity, and there have been no realistic implementations.

This article aims to overcome all these previous LSOD obstacles. It demonstrates that LSOD can be checked naturally using *Program Dependence Graphs* (PDGs) for concurrent programs. PDGs have already been developed as an IFC analysis tool for full sequential Java [16, 15,46,51], and demonstrated high precision and scalability. The current work shows how to use PDGs for a precise LSOD checker. It uses a new definition of low-equivalent traces, which – in case of nonterminating traces – avoids certain problems of earlier definitions. Exploiting the structure of PDGs, the algorithm is flow-sensitive, object-sensitive, and context-sensitive. It is sound and does not impose restrictions on the thread or program structure. It also relaxes the classical LSOD definition by allowing secure nondeterminism, while preserving soundness. It turns out that flow sensitivity is the key to eliminating soundness gaps and unrealistic restrictions. The algorithm also exploits advances in the may-happen-in-parallel (MHP) analysis of concurrent programs, which allow even time-sensitive MHP and IFC.

Note that PDG-based IFC for sequential programs has been described in detail in [16]; we assume some familiarity with this earlier work. The current article concentrates on PDG-based IFC for concurrent programs. We present an informal overview (section 2), formally develop the new RLSOD criterion and its soundness proof (section 3), summarize PDGs for concurrent programs (section 4), show how the criterion can be precisely and soundly approximated by a PDG-based static RLSOD check (section 5), explain the algorithm details (section 6), and present data about performance and precision (section 7). Related work is discussed in section 8.

The JOANA tool. PDG-based IFC, including the new algorithm described in this article, has been fully implemented. The system, called JOANA, is available for public download, or can be used by everybody through a Java webstart GUI.[1] The engineer must provide Java sources to be analysed, where all input and output statements are annotated "high" or "low" (other statements do not need annotations). JOANA can handle full Java bytecode with arbitrary threads, scales to ca. 50kLOC, and empirically demonstrates high precision [16,15,46, 14]. JOANA is based on a stack of sophisticated program analysis algorithms (pointer analysis, exception analysis, PDG construction; some details are described in section 4). JOANA minimizes false alarms through flow-, context-, object-, and field-sensitive analysis techiques. JOANA allows declassification along sequential information flows. In concurrent programs, all possibilistic and probabilistic leaks are discovered. JOANA and the underlying program analysis was developed over the last 15 years, and was used in realistic case studies such as [25]. The practical application is described in detail in [12].

```
if (h==1)
  l = 42
else
  l = 17;
... // the following assignment can be
... // far away from the IF; as long as it
... // postdominates the IF and there is no
... // intermediate output of h or l,
... // the program is secure
l = 0;
print(l);
```

```
h = 1;
l = 2;
x = f(h);
y = f(l);
print(y);

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

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

Fig. 2 Three secure program fragments. Flow- or context-insensitive analysis will generate false alarms.

2 Overview of approach

Security policy. IFC analysis must discover all possible violations of confidentiality and integrity – including probabilistic ones – for realistic programs. Our IFC analysis thus aims to provide a sound, precise, scalable and non-restrictive noninterference criterion for programs in full Java containing arbitrary threads; needing few annotations, and admitting possible declassifications.

Why LSOD? We chose LSOD as the fundamental mechanism because it has the huge advantage of being scheduler-independent. However, attempts to define LSOD in a termination-sensitive way led to severe restrictions (see below). We therefore aim for a definition which is termination insensitive, but flow-, context-, and object-sensitive. From a practical viewpoint, we believe that these features are more important than termination

[1] `joana.ipd.kit.edu` provides download, webstart application, and other information

sensitivity; they also overcome previous obstacles to the use of LSOD. In particular, our new RLSOD criterion will not prohibit secure low-nondeterminism.

Flow sensitivity. A flow sensitive analysis takes statement order into account, and a context-sensitive analysis takes procedure calling context into account. PDG-based IFC was introduced because PDGs are naturally flow- and context-sensitive. Studies have shown that this reduces false alarms considerable (e.g., [15,17]); figure 2 left presents two sequential examples (we assume `h` to contain a high value and `l` to contain a low value). For object-oriented programs, object- and field-sensitivity are similarly important [16].

Now consider the multi-threaded example in Figure 2 right. A flow-insensitive IFC analysis ignores statement order, thus assumes that lines 8 and 9 are interchangeable, and generates a false alarm. We will return to this example in sections 3 and 8. Note that typical security type systems are flow- and/or context-insensitive and will reject all programs in Figure 2. The type system in [22] is flow-sensitive, but not context-sensitive.

Classification of statements and data. Program data and operations are classified either *low* (public) or *high* (secret).[2] Note that in our flow-sensitive approach, classification of values and variables happens *per statement or expression* in the source code – there is no global classification of variables. Technically, PDG nodes n (in particular nodes for (sub)expressions in statements) are statically classified: $cl(n) = L$ resp. $cl(n) = H$. It is enough to classify inputs (sources) and outputs (sinks), as the classification of all intermediate nodes can easily be computed by a fixpoint iteration on the PDG [16]. Hence a variable may at one program point (PDG node) contain a low value, and at another point a high value, as both variable occurrences are represented by different PDG nodes. Still, soundness is guaranteed, while flow sensitivity (which is naturally provided by PDGs) offers precision gains and fewer restrictions on programs.

We further assume that program input and output consist of streams of (perhaps non-primitive) values, where a complete stream has a security classification. In accordance with flow sensitivity, high input streams are *not* part of initial memory, but all streams (except `stdin` and `stdout`) have to be explicitly opened.[3] Inputs are *low-equivalent* if they coincide on low input streams.

Attacker model. We assume that an attacker knows the source code, and can observe execution of all operations (i.e. dynamically executed statements) and their operands that are classified low. However, the attacker cannot observe high operations or high operands. For example, if the source statement `print(stream,x)` is classified low, then its dynamic execution outputs a value on a low output stream, which can be observed; if `read(stream,x)` is classified high, the dynamically read value of `x` comes from a high input stream and cannot be observed.[4] As explained, variables do not possess a global classification; correspondingly, the attacker *cannot* see all low values at any time.

```
void main():
  x = inputPIN();
  while (x > 0)
    print("x");
    x--;
  while (true)
    skip;
```

```
void main():
x=inputPIN();
while (x>0)
  print ("x");
  x--;
  if (x==0)
    {while (true)
       skip;}
```

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

```
void main():
  x = inputPIN();
  while (x == 0)
     skip;
  print("x");
  while (x == 1)
     skip;
  print("x");
  ...
  while (x == 42)
     skip;
  print("x");
  ...
```

Fig. 3 Four tough nuts for termination-insensitive definitions of low-equivalent traces. All programs contain termination leaks and gradually leak (part of) the PIN.

We assume that the attacker *can* distinguish the relative order of reads/writes to different variables (in contrast to [53], see discussion in section 8). We further assume the attacker *cannot* observe whether a program is in an infinite loop. But the attacker may have knowledge about the probabilities $P_i(r)$ of input i causing a certain low-observable behaviour r.

Low-equivalent traces. The definition of LSOD is based on low-equivalent traces. A trace of a program execution is a (possibly infinite) list of program configurations, where a configuration includes the executed operation, the memory before execution, and the memory after execution. Note that a trace is valid only with respect to a specific interleaving of program threads. Low-observable events are configurations from a trace which read or write low values; only memory cells which are read or written by a low operation are part of low-observable events. The *low-observable behaviour* of an execution trace is the subtrace which contains only low-observable events. As the attacker knows the source code and PDG, the attacker can reconstruct from the low-observable behaviour which PDG node caused a low-observable event.

Low-equivalent traces have identical low-observable behaviour. LSOD demands that *any two executions with low-equivalent inputs have low-equivalent traces.* Thus LSOD is defined similarly to classical (sequential) noninterference, using traces instead of program states. This natural definition, as formalised in the literature, how-

[2] The implementation can handle arbitrary security lattices.

[3] reads and writes on unopened streams are assumed to throw an exception, and PDGs can handle exceptions precisely [16,14].

[4] If `read(stream,x)` is classified low, but `stream` is classified high, the resulting explicit illegal flow is trivially discovered in the PDG [16]. Similarly if `print(stream,x)` is classified high, but `stream` is classified low.

ever allows one trace to terminate and the other to not terminate. Hence the problem of *infinite traces* and *termination channels* needs to be discussed before we formally present the new definition of low-equivalent traces.

Termination leaks. Consider the top right program in Figure 3, whose input in line 2 is high data and whose `print`-statement is low-observable. If a run of the program does not terminate, the `print`-statement is delayed infinitely, which leads to the conclusion that the input was < 0. Worse, Figure 3 bottom right exploits a termination channel that leaks the PIN completely. The two left programs behave identically to the bottom right program, and also contain termination channels, but contain an additional implicit flow (the right programs do not contain implicit flow).[5] It is characteristic for termination leaks that the attacker must know that a program loops, in order to exploit the observable behaviour. It is known that in interactive programs, termination channels can leak arbitrary amounts of information [2].

To prevent termination channels, several variants of LSOD and PN *forbid low-observable events behind loops guarded by high data.* Such algorithms will disallow the programs in Figure 3. However, in practice this is an unacceptable restriction. Sometimes, program analysis can deduce that a loop will terminate, and the restriction can be relaxed. But in general, no other means to always avoid termination leaks are known. Therefore several authors – including ourselves – allow termination channels (see also section 8). Hence our attacker model assumes that the attacker cannot observe nontermination.

We thus aim at a sound definition of LSOD which however may allow termination leaks. This approach has already been tried in earlier research – in particular in [47,53]. Their LSOD definitions permit termination channels, and declare traces to be low-equivalent if their low-observable behaviour is equal up to the length of the shorter sequence of low-observable events. But as pointed out in [21], this may lead to unintended leaks. Consider the program in Figure 3 top left, whose traces always diverge, and assume that the input PIN is high data and that the print statement is low-observable. The program exposes the input PIN by printing an equal number of x's to the screen. If low-equivalence of traces is confined to the length of the shorter sequence of low-observable events, this behaviour is perfectly legal, because all traces with low-equivalent inputs are equal up to the length of the shorter sequence.

The new approach. To solve this problem, we suggest the following new definition of low-equivalent traces: For finite traces, we stick to the common definition that they are low-equivalent if their low-observable behaviours are equal. If one trace is finite and the other is infinite, then the finite trace must have *at least as many* low-observable events as the infinite one, the low-observable behaviours must be equal up to the length of the shorter sequence, *and* the low-observable events missing in the infinite trace must be missing due to nontermination. The latter constraint is new. It means that *prior* to the missing events, a condition e.g., in an IF ("branching point") must have been evaluated which dynamically led into an infinite loop. Thus the new constraint makes sure that the missing events leak information only via termination channels.

Of course, for two infinite traces with infinite low-observable subtraces, these subtraces must coincide completely. The new definition of low-equivalent traces is soundly approximated through PDGs and slicing, resulting in a precise analysis for probabilistic noninterference. Additional exploitation of the concurrent control flow graph prevents rejection of secure low-nondeterminism.

3 Formalizing low-equivalent traces and LSOD

In the following, we will formally develop the definition of the noninterference criterion, prove that it guarantees LSOD, and use it as basis for an algorithm. To begin with, let us repeat the original PN definition [44,42]:

Definition 1 (Probabilistic noninterference) A program satisfies probabilistic noninterference if for all pairs (t, u) of low-equivalent inputs the following holds:

Let Θ be the set of possible program runs (traces) resulting from t and u. For each $T \in \Theta$, the following must hold: Let $\mathfrak{T}$ be the set of program runs possibly caused by t that are low-equivalent to T. Let $\mathfrak{U}$ be the analogous set for u. Then $\sum_{r\in\mathfrak{T}} P_t(r) = \sum_{r\in\mathfrak{U}} P_u(r)$ must hold, where $P_i(r)$ is the probability of trace r under input i.

Note that in practice the $P_i(r)$ can be scheduler-dependent and are very difficult to compute or measure; worse, the number of summands might be infinite. Fortunately we will later see that no values for the $P_i(r)$ are needed for LSOD. Note also that any possibilistic leak is a probabilistic leak as well, but not vice versa.

To illustrate definition 1, consider the program in Figure 10 (from [44]). It reads a `PIN` and employs three threads to compute a value `result`, which is finally printed. There is no explicit or implicit flow from `PIN` to `result`. But if the input `PIN` is less than twice the value of variable `mask`, then `PIN`'s value is copied bit-wise into `result` and printed (provided that scheduling is starvation-free). Thus for some interleavings the secret PIN influences the probability of public output, and definition 1 is violated.[6] It has been argued that the leak in Figure 10

[5] The bottom left program was proposed by an anonymous reviewer of a previous version of this work.

[6] Formally, definition 1 is violated as follows. There is only high input, so all inputs are low-equivalent. Let $t = 42, u = 17$, which are both $< 2 \cdot$ `mask`. Then there is a trace T which prints 42. Let $\mathfrak{T}$ be all traces caused by t low-equivalent to T, let $\mathfrak{U}$ be all traces caused by u low-equivalent to T. In particular $T \in \mathfrak{T}$, so all $r \in \mathfrak{T}$ will print 42. Thus $\sum_{r\in\mathfrak{T}} P_t(r) \geq P_t(T) > 0$. But $\mathfrak{U} = \emptyset$ because all traces caused by u print 17. Thus $\sum_{r\in\mathfrak{U}} P_u(r) = 0 \neq \sum_{r\in\mathfrak{T}} P_t(r)$.

```
1 void main():
2    fork thread_1();
3    l = 0;
4    print(l);

5 void thread_1():
6    h = inputPIN();
7    while (h != 0)
8       h--;
9     l = input();
10   print(l2);
```

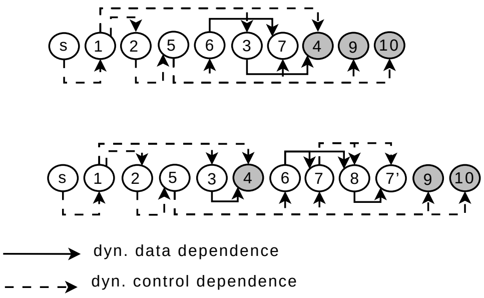


Fig. 4 A program and two possible traces. The first trace results from input (inputPIN() = 0, input() = 0), the second from (inputPIN() = 1, input() = 0). The shaded nodes represent the low-observable behaviour.

is in fact possibilistic; to make it truly probabilistic one might e.g., add a random-based IF to thread `Beta` which assigns `result` in only 80% of all runs.

3.1 Traces and dynamic dependences

Definition 2 (Trace) An *operation* is a dynamic instance of a program statement (e.g., assignment execution, procedure call, thread fork). For operation o, $stmt(o)$ is the corresponding source program statement.

A *trace* is a list of events of the form $(\overline{m}, o, m)$, where o is an operation, $\overline{m}$ is the memory before execution of o, and m is the memory after execution of o.

The *low-observable behaviour* of a trace

$$T = (\overline{m}_1, o_1, m_1), \ldots, (\overline{m}_k, o_k, m_k)$$

is a list of events

$$obs_{low}(T) = evt_{low}(\overline{m}_1, o_1, m_1), \ldots, evt_{low}(\overline{m}_k, o_k, m_k)$$

where

$$evt_{low}(\overline{m}, o, m) = \begin{cases} (\overline{m}\mid_{use(o)}, o, m\mid_{def(o)}) & o \text{ reads or writes low values;} \\ \lambda & \text{otherwise.} \end{cases}$$

$def(o)$ are the variables defined (written) in o, while $use(o)$ are the variables used (read) in o. $\overline{m}\mid_{use(o)}$ resp. $m\mid_{def(o)}$ are the memory cells in m resp. $\overline{m}$ which are used resp. defined by o; λ is the empty event.

We write $o \in T$ iff $\exists \overline{m}, m : (\overline{m}, o, m) \in T$, and $LO(o)$ iff for $o \in T$: $evt_{low}(\overline{m}, o, m) \neq \lambda$. For $o, o' \in T$, we write $o < o'$ if o is executed before o': $o = o_i, o' = o_j, i < j$.

It is important to note that a trace includes *all* operations of a specific program execution with specific input; due to interleaving, many traces for the same input may exist. Operations in (different) traces can be uniquely identified by their calling-context and control-flow history (see below). Operations which read or write low values are low-observable, together with the corresponding parts of the memory. Thus only $\overline{m}\mid_{use(o)}$ resp. $m\mid_{def(o)}$ are low-observable.

To better understand the latter fact, remember that due to flow sensitivity, there is no fixed separation into low and high memory. $\overline{m}\mid_{use(o)}$ and $m\mid_{def(o)}$ are exactly all the low cells touched by o; this "operation-wise exact fit" has the effect to weaken the requirements for low equivalence while maintaining soundness (see below), ultimately reducing false alarms. A global low memory M_{low} would render *less* traces low equivalent (i.e. more false alarms), as $\overline{m}\mid_{use(o)}, m\mid_{def(o)} \subseteq M_{low}$. This subtle insight is another explanation why flow-sensitive IFC is more precise.

Example. Figure 4 presents examples for traces and their low-observable behaviour. Remember that only input and output are explicitly classified as low or high, and that memory cells are not globally classified.

For later use with the LSOD criterion, the definition of traces is enriched with dynamic control and data dependences as in Figure 4: these connect dynamic reads and writes of variables (with no intermediate writes to the variables), and dynamic conditions from if, while etc. and the operations "governed" by these conditions, This is formalised in

Definition 3 (Dynamic dependences) Let T be a trace.

1. Operation $o \in T$ is dynamically control-dependent on operation $b \in T$, written $b \overset{dcd}{\dashrightarrow} o$, iff
   - o is a thread entry and b is the corresponding fork operation, or
   - o is a procedure entry and b is the operation that invoked that procedure, or
   - b is the director of the innermost control region of o.[7]
2. Operation o is dynamically data dependent on operation a in T, written $a \overset{v}{\dashrightarrow} o$, iff there exists a variable $v \in use(o) \cap def(a)$, $a < o$, and there is no operation $o' \in T$, $v \in def(o')$ where $a < o' < o$.
3. $DFS_T(o) = \{q \in T \mid o\ (\overset{v}{\dashrightarrow} \cup \overset{dcd}{\dashrightarrow})^*\ q\}$ denotes the set of all operations that are (transitively) dynamically control or data dependent on o.
   $DCD_T(o) = \langle start, q_2, \ldots, q_{n-1}, o \mid q_i \in T, q_i < q_{i+1}, q_i \overset{dcd}{\dashrightarrow}{}^* o\rangle$ is the list of operations on which o is (transitively) dynamically control (but not data) dependent.

If T is fixed, we just write DFS and DCD. Note that dynamic dependencies are cycle-free. $DFS(o)$, the operations potentially influenced by o, can be seen as a dynamic forward slice; DCD can be seen as a dynamic backward control slice (cmp. section 4). Every operation

[7] that is, b is a branching point with immediate postdominator $PD(b)$ and $b < o < PD(b)$ [52].

has exactly one predecessor on which it is dynamically control-dependent (except the *start* operation, which has no dynamic predecessor). For $o \neq start$ and $b \overset{dcd}{\dashrightarrow} o$, b is the unique dynamic control predecessor of o, written $b = dcp(o)$.

Example. In Figure 4 (lower part), $DFS(5) = \{6, 7, 8, 7', 9, 10\}$, and $DCD(4) = \langle start, 1, 4\rangle$.

3.2 Infinite delay and low-equivalency

We are now ready to tackle low-equivalency of traces. As explained earlier, we want to define low-equivalency of two traces T, U such that for *infinite* T, if T misses a low-observable operation o executed in U, o is missing due to an infinite delay in T. Other reasons for non-execution of o in T are not allowed.

This idea requires a formalisation of the notions of "infinite delay" and "an operation happens in two different traces". First we observe that an operation is uniquely identified by its calling context and control flow history. More formally, $p = q$ holds for operations $p \in T$ and $q \in U$ if $stmt(p) = stmt(q)$, and either $p = q = start$, or $dcp(p) = dcp(q)$. This recursive definition terminates as backward control dependency chains are finite. Thus $p = q \iff DCD(p) = DCD(q)$. This definition explicitly includes the case that an operation occurs in two different traces T and U, written $o \in T \cap U$. Note that $o \in T \cap U$ still allows that the memories (in particular the high parts) in both traces at o are not identical.

Next we observe that the execution of a branching point in a trace T triggers the execution of *all* operations in the chosen branch which are dynamically control-dependent on the branching point; up to the next branching point. For example, in the code fragment `if(b){o1; o2}`, both `o1` and `o2` are (statically and dynamically) control-dependent on `b` (but `o2` is not control-dependent on `o1`). If `b` evaluates to `true` and the then-branch is executed, *both* `o1` and `o2` are executed, unless `o1` does not terminate.[8]

In terms of traces, if $b_1 \overset{dcd}{\dashrightarrow} o_1, o_2 \dots o_k \overset{dcd}{\dashrightarrow} b_2$ (where not necessarily $o_i \overset{dcd}{\dashrightarrow} o_{i+1}$), and $o_1 \in T$ (that is, o_1 belongs to the branch chosen by b_1 and thus is executed) then $o_2, \dots o_k$ are executed as well, *unless* there is nontermination in some o_i, causing o_{i+1} to be delayed infinitely. Other possibilities for the non-execution of o_{i+1} do not exist, because control dependency by its very definition implies that b_1 (and nobody else) decides about the execution of $o_1 \dots o_k$. The same argument applies if b occurs in two traces T and U. Hence we define

Definition 4 (Infinite delay) Let T, U be traces and let both execute branching point b: $b \in T \cap U$. Let $o \in T$ be an operation where $b \overset{dcd}{\dashrightarrow} o$ (thus o belongs to the branch b chooses to execute in T). If $o \notin U$, U infinitely delays o.

Thus if both T and U execute b and choose the same branch, then either both T and U execute o, or U does not execute o due to an infinite loop between b and o. This definition is used for the formalization of low-equivalent traces:

Definition 5 (Low-equivalence of traces, $\sim_{low}$) Let T and U be two traces. Let $obs_{low}(T) = (\overline{m}_0, o_0, m_0)\cdots$ and $obs_{low}(U) = (\overline{n}_0, q_0, n_0)\cdots$ be their low-observable behaviours. Let k_T be the number of events in $obs_{low}(T)$ and k_U be the number of events in $obs_{low}(U)$. T and U are low-equivalent, written $T \sim_{low} U$, if one of the following cases holds:

1. T and U are finite, $k_T = k_U$, and $\forall 0 \leq i < k_T : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$
2. T is finite and U is infinite, and
   - $k_T \geq k_U$,
   - $\forall 0 \leq i < k_U : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, and
   - $\forall k_U < j < k_T$: U infinitely delays an operation $o \in DCD(o_j)$.
3. T is infinite and U is finite, and
   - $k_U \geq k_T$,
   - $\forall 0 \leq i < k_T : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, and
   - $\forall k_T < j < k_U$: T infinitely delays an operation $o \in DCD(q_j)$.
4. T and U are infinite, and
   - if $k_T = k_U$, then $\forall 0 \leq i < k_T : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$.
   - if $k_T > k_U$, then $\forall 0 \leq i < k_U : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, and $\forall k_U < j < k_T$: U infinitely delays an operation $o \in DCD(o_j)$.
   - if $k_T < k_U$, then $\forall 0 \leq i < k_T : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, and $\forall k_T < j < k_U$: T infinitely delays an operation $o \in DCD(q_j)$.
   - if $k_T = k_U = \infty$, then $\forall i : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$.

In fact the definition can be expressed in a more compact form:

Observation. $T \sim_{low} U \iff$

- $\forall 0 \leq i \leq \min(k_T, k_U) : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, and
- if $k_T \neq k_U$, and w.l.o.g. $k_T > k_U$, then U is infinite and $\forall k_U \leq j < k_T$: U infinitely delays an operation $o \in DCD(q_j)$.

We will however refer to cases 1 – 4 in the following text. If all programs terminate, definition 5 reduces to case 1, which does not rely on *DCD* or infinite delay. In case of nontermination, i.e. infinite traces, all low-observable o_j missing in the shorter trace must be

[8] Note that exceptions and handlers generate additional control dependencies in PDGs and traces [16]. Thus if `o1` may throw an exception, the dependency situation is more complex than in a "regular" `if(b){o1;o2}`. Still, the subsequent argument for traces holds.

missing due to infinite delay. "U infinitely delays $o \in DCD(o_j)$" expresses that the delayed operation o must not necessarily be o_j itself, but can be a dynamic control predecessor of o_j. In any case, o is on the dynamic control path between o_{k_U-1} and o_j, and the infinite loop in the shorter trace is *before* o: according to definition 4, there must be a branching point $b \in T \cap U$, where $b \overset{dcd}{\dashrightarrow} o, o \notin U$.

Interestingly, the essence of definition 5 can be expressed without infinite delay.[9] Imagine every program has an additional `print( 'done')` as the very last statement. Then the condition "U infinitely delays an operation $o \in DCD(o_j)$" implies "`print('done')`$\notin U$". Perhaps it would be possible to re-formulate definition 5 accordingly (thus replacing "U infinitely delays ..." by "`print('done')`$\notin U$") and construct an alternate soundness proof. In any case, *DCD*, *DFS* will be needed for the proof.

Note that definition 5 is stronger than the low-equivalency definitions in [53,21]: the latter authors demand low equivalence not for T and U, but only for all subtraces of individual variables. This weaker definition results in more traces being low equivalent; it however assumes that the attacker cannot distinguish between relative access order for different variables. It also assumes that variables are globally classified, which in our flow-sensitive setting they are not. Section 8 compares both definitions in detail.

The following definition of LSOD is standard, but uses the new definition of low-equivalent traces:

Definition 6 (Low-security observational determinism) A program is low-security observational deterministic if the following holds for every pair (t, u) of low-equivalent inputs: Let $\mathfrak{T}$ and $\mathfrak{U}$ be the sets of possible traces resulting from t and u. Then $\forall\ T, U \in \mathfrak{T} \cup \mathfrak{U}$: $T \sim_{low} U$ must hold.

Note that we will later relax this definition and allow secure low-nondeterminism while maintaining soundness. We will now discuss definitions 5 and 6 using the examples from Figures 2 and 3.

Example 1. For a simple multi-threaded program without termination leaks, consider Figure 2 right. `print(l)` is explicitly classified low, and `inputPIN` is explicitly classified high. Remember that `l` and `h` are not globally classified; only input/output is explicitly classified. For all other statements the classification is computed through a fixpoint iteration on the PDG.[10] In particular, if $x \in BS(y)$, $cl(x) \sqsubseteq cl(y)$ must hold, where $Low \sqsubset High$ [16]. Thus `h` in line 6 is high, `l` in line 5 is low, as `inputPIN` $\in BS(\mathtt{h}_6)$ and $\mathtt{l}_5 \in BS($`print(l)`$)$. `l=h` is however *not* classified low, because `l=h` $\notin BS($`print(l)`$)$ (see the PDG in figure 6).

Next we observe that all inputs are low-equivalent, as the only input is high. Now let T, U be two possible traces for different `inputPIN`s (in this example, several traces exist for the same input due to interleaving). Hence `l=0`, `print(l)` are the only low-observable operations in T and U. The operation order may however be different for T and U due to interleaving. Hence $T \not\sim_{low} U$. The example demonstrates the central weakness of LSOD: even secure low-determinism is prohibited. The new RLSOD criterion however accepts the example (see below). But RLSOD only works because flow sensitivity of BS prevents `l=h` to be low-observable, which otherwise would cause a false alarm.

Example 2. To understand the treatment of termination leaks, consider the top left program in Figure 3. Again all inputs are low-equivalent. We show that the program does not fulfill definition 5. Obviously the program diverges for any `inputPIN`, but for different `inputPIN`s prints different numbers of `"x"`s. Let us select trace T for one `inputPIN` n and trace U for a different `inputPIN` m. Then we are in the infinite/infinite case of definition 5. T contains n instances of `print("x")`, as well as n instances of `while(x>0)` and n instances of `x--`; thus $k_T = 3n$. U contains m instances `print("x")` etc. W.l.o.g., assume $k_T > k_U$. The low-observable o_j are the `print("x")` in T which are missing in U. $DCD(o_j)$ is the list of operations $q_i \in T$ on which $o_j \in T$ is dynamically control-dependent, namely the list of dynamically executed loop tests:
$DCD($`print("x")`$) = \langle$`while(x>0), while(x>0), ...`$\rangle$.

Definition 5 demands that U infinitely delays an operation o from this list: there must be an infinite loop *before* o. But this is not the case! According to definition 4, it would not only require $o \notin U$ (which is the case for the first `while(x>0)` not in U), but that there is a branching point $b \in T \cap U, b \overset{dcd}{\dashrightarrow} o$ which selects the same branch in both traces, but never proceeds to o in U.

Such a b does not exist. Indeed the operation causing the infinite delay, namely `while (true)`, is not between any b and o, but at the end of the program. Thus $T \not\sim_{low} U$. Hence this program is not LSOD; it violates definition 6. In this example, definition 5 discovered the termination leak, but in general it will not discover termination leaks – as in the next example.

Example 3. The bottom left program in Figure 3 fulfills definition 5 as follows. Let T, U, k_T, k_U, o_j as above. Thus the first low-observable operation $o_j \in T$ missing in U is occurence no. $k_U + 1$ of `print("x")`. Again o_j is dynamically control-dependent on some $o =$ `while(x>0)`. But this time, there is $b \in T \cap U$, namely occurence $k_U - 1$ of `if(x==0)`, which dynamically controls the k_U'st `while(x>0)` in T. But the latter is not in U any more. Thus $b \overset{dcd}{\dashrightarrow} o$, and the infinite loop indeed happens between b and o. Hence $T \sim_{low} U$. That is, by definitions 5 and 6, the program is wrongfully declared secure – which is ok, as we exclude termination leaks.

[9] We thank one reviewer for observing this.

[10] PDGs and the backward slice BS are explained in detail in section 4; here we rely on some preliminary understanding.

We see that definitions 5 and 6 may miss termination leaks – which is a design feature. But sometimes we are lucky, and termination leaks are discovered anyway, as in example 2.

3.3 Soundness of LSOD criterion

Soundness means that, under the conditions of the following theorem, all probabilistic leaks are discovered by definition 6. Precision means that the number of false alarms is small. In this section we outline the soundness proof. Precision is investigated in section 7.

Zdancewic and Myers [53] observed that probabilistic leaks can only occur if the program contains concurrency conflicts such as data races. LSOD is guaranteed if there is no implicit or explicit flow, and in addition program parts influencing low-observable behaviour are conflict free. Their observation served as a starting point for our own work, as we realised that not only explicit and implicit flow can naturally be checked using PDGs, but also conflicts and their impact are naturally modelled in PDGs enriched with conflict edges. We thus provide the following definition:

Definition 7 (Data and order conflicts) Let a and b be two operations that may happen in parallel.

- There is a data conflict from a to b, written $a \overset{dconf}{\rightsquigarrow} b$, iff a defines a variable v that is used or defined by b.
- There is an order conflict between a and b, written $a \overset{oconf}{\leftrightsquigarrow} b$, iff both operations are low-observable: $LO(a) \wedge LO(b)$.
- An operation o is *potentially influenced by a data conflict* if there exist operations a, b such that $o \in DFS(b)$ and $a \overset{dconf}{\rightsquigarrow} b$.

The following lemma states that in all possible traces which are produced by a set of low-equivalent inputs, the operations which are not influenced by high data or by execution order conflicts are identical "modulo termination". In particular, for terminating traces these operations are completely identical.

Lemma 1 *Let p be a program. Let T be a trace of p and Θ be the set of possible traces whose inputs are low-equivalent to the one of T. Let o be an operation of p that is not potentially influenced by a data conflict $a \overset{dconf}{\rightsquigarrow} b$ or by an operation q reading high input: $o \notin DFS(a) \cup DFS(b) \cup DFS(q)$. If $o \in T$, then every $U \in \Theta$ either executes o or infinitely delays an operation in $DCD(o)$.*

The proof is in appendix A; the full proof can be found in [8]. Without the dependency machinery from definition 5, the proof would not be possible. From the lemma it is a small step to the statement that low-equivalent inputs generate low-equivalent traces. This fundamental soundness theorem can now be stated:

Theorem 1 *A program is low-security observational deterministic according to definition 6 if for all traces T and all operations $o, o', o'' \in T$:*

1. *no low-observable operation o is potentially influenced by an operation reading high input:*
$$LO(o) \wedge \neg LO(o') \implies o \notin DFS_T(o')$$
2. *no low-observable operation o is potentially influenced by a data conflict:*
$$LO(o) \wedge o' \overset{dconf}{\rightsquigarrow} o'' \implies o \notin DFS_T(o') \cup DFS_T(o'')$$
3. *there is no order conflict between low-observable operations:*
$$\neg(o \overset{oconf}{\leftrightsquigarrow} o')$$

Proof. see Appendix A; the full proof is in [8].

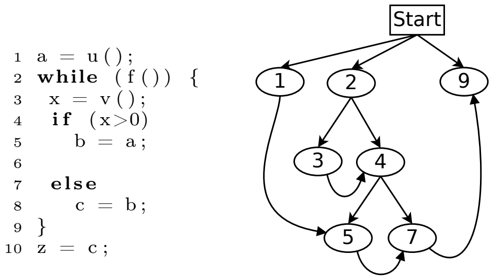
In the criterion, the first rule ensures that the implicit and explicit flow to o does not transfer high data. The second rule ensures that high data cannot influence the data flowing to o via interleaving. The third rule ensures that high data cannot influence the execution order of low-observable operations via interleaving. Note that the theorem is only valid if *sequential consistency* in the sense of the Java memory model can be assumed. The Java memory model was designed to guarantee sequential consistency for race-free programs, and formal definitions plus machine-checked guarantees of the JMM are available [28, 27].

Theorem 2 *If a program is LSOD according to definition 6, it is probabilistically noninterferent according to definition 1.*

Proof. see appendix A. The proof is in a sense trivial, as the sum of probabilities for all traces possibly generated by two low-equivalent inputs must equal 1 (which also explains why LSOD does not need explicit probability values for traces). Note that for two low-equivalent inputs more than one trace may result, because our algorithm – as explained below – does not prohibit useful nondeterminism, as long as soundness is maintained.

Corollary 1 *LSOD, as defined in Definition 6, is scheduler-independent.*

Proof. Using the terminology of Definition 1, scheduler behaviour will influence the probability distributions for certain interleavings and thus the $P_t(r)$ resp. $P_u(r)$. The proof of theorem 2 however demonstrated that under LSOD, the *sum* of these probabilities is always 1. Only these sums are needed to guarantee probabilistic noninterference, thus the latter is invariant under the probability distribution of interleavings.



```
1 a = u();
2 while (f()) {
3   x = v();
4   if (x>0)
5     b = a;
6
7   else
8     c = b;
9 }
10 z = c;
```

Fig. 5 A small program and its dependence graph

4 Dependence Graphs and Noninterference

In the following, we will present a static analysis which soundly approximates the above LSOD criterion. This analysis is based on program dependence graphs (PDGs). In this section, we present the necessary facts about PDGs for multi-threaded programs, which are then exploited in sections 5 and 6.

4.1 PDGs for Sequential Programs

Program dependence graphs are a standard tool to model information flow through a program. In an (intraprocedural) PDG $G = (N, \to)$, N comprises program statements or expressions. There are two kinds of edges: *data dependences* and *control dependences*, thus $\to = \overset{D}{\to} \cup \overset{C}{\to}$. Dependence $x \overset{D}{\to} y$ means that statement x assigns a variable which is used in statement y, without being reassigned on any control flow path from x to y. Dependence $x \overset{C}{\to} y$ means that the mere execution of y depends directly on the value of the expression x (which is typically a condition in an if- or while-statement, see e.g., [24] for formal definitions). A path $x \to^* y$ means that information can flow from x to y; if there is no path, it is guaranteed that there is no information flow [18,38,36, 51]. Exploiting this fundamental property, all statements possibly influencing y (the so-called *backward slice*) are easily computed as $BS(y) = \{x \mid x \to^* y\}$. y is called the *slicing criterion* of the backward slice. Similarly, the forward slice is $FS(x) = \{y \mid x \to^* y\}$.

As a small example, consider the program and its dependence graph in Figure 5 (from [16]). Control dependences are shown as straight edges, data dependences are shown as curved edges. There is a path from statement 1 to statement 9 (i.e. $9 \in FS(1)$), indicating that input variable `a` may eventually influence output variable `z`. Since there is no path $(1) \to^* (4)$ $(1 \notin BS(4))$, there is definitely no influence from `a` to `x>0`.

The Slicing Theorem [18,38] states that for any terminating execution reaching statement x, the program and $BS(x)$ compute the same sequence of values for each variable used in x. Thus a correct PDG may have too many edges, but never too few ("soundness"). But of course, PDGs should have as few edges as possible ("precision"). Note that due to decidability problems, "complete" precision can never be achieved while maintaining soundness. Section 5 contains formal soundness properties of slices, as exploited by our static LSOD check.

PDGs and slices for languages with procedures, exceptions, pointers, objects, arrays etc. are much more complex. Interprocedural analysis is typically based on the context-sensitive Horwitz-Reps-Binkley (HRB) algorithm [37,19], which uses *summary edges* to model flow through procedures.[11] Full sequential Java requires even more complex algorithms, in particular for pointer analysis. The pointer analysis used in JOANA is object-sensitive, field-sensitive, and optionally flow-sensitive; it is described in [11,16]. Precise pointer analysis is a prerequisite for precise treatment of dynamic dispatch in object-oriented languages [16]. Another issue are exceptions, which may cause a lot of additional control flow. Precise treatment of exceptions in PDGs is described in [16,14]. Thus in general, computing $BS(x)$ involves more than just backward paths in the PDG.

In-depth descriptions of slicing techniques can be found in [24]. Note that the slicing theorem has been shown for all the extensions mentioned above – and the soundness of the current work only depends on the slicing theorem, not on the specific variant of slicing or pointer analysis.

Of course, slicing precision directly influences IFC precision and false alarms. In the early days of PDGs and slicing, often a backward slice comprised almost the complete program. Today, PDGs have become much more precise since they are flow-sensitive, context-sensitive, and object-sensitive: the order of statements is taken into account, as is the actual calling context for procedures, and the actual reference object for method calls. Thus the backward slice never indicates influences that are in fact impossible due to the given statement execution order or call stack of the program; only so-called "realizable" paths are considered (that is, paths dynamically possible with respect to call stack and statement order). As a consequence, slice size has dropped dramatically (see e.g., [3,4]). But this precision is not for free: interprocedural PDG construction can have complexity $O(|N|^3)$, object-sensitive pointer analysis is similarly expensive, and flow-sensitive pointer analysis is so expensive that it is activated on demand only. In practice, PDG use is limited to programs of about 100kLOC [4]. Section 7 presents data about precision and scalability of our PDG construction algorithm.

4.2 Noninterference and PDGs

As mentioned, $a \notin BS(b)$ guarantees that there is no information flow from a to b. This is true for all information flow which is not caused by hidden physical side

[11] HRB use so-called system dependence graphs, which in this article are subsumed under the PDG notion.

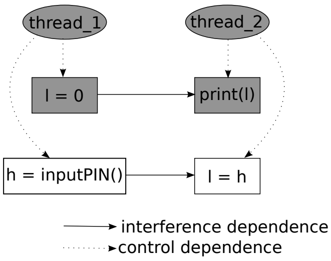


Fig. 6 PDG for example in Figure 2 right. $BS(\mathtt{print(l)})$ is shaded.

channels such as timing leaks. It is therefore not surprising that slicing is a natural tool for IFC [45,1]. Illegal flow becomes visible as a PDG path from a secret value to a public variable. In Figure 5, assume `x` is secret and `z` is public; the PDG path $3 \to^* 9$ (i.e. $3 \in BS(9)$) exposes the illegal (implicit) flow from `x` to `z`. In Figure 7, slicing uncovers an illegal flow in a multi-threaded example (see next section).

More examples for PDG-based IFC can be found in [16,8]. In 2009, we provided a machine-checked proof that (sequential) noninterference holds if no high variable or input is in the backward slice of a low output. This result was shown for the intraprocedural as well as the interprocedural (HRB) case [51,49].

Let us finally mention that – due to potentially non-local effects of any points-to, alias, or dependency relation – the PDG of a complete system cannot be obtained by just combining PDGs of subsystems. PDGs thus require a whole-program analysis, and all library functions have to be analysed together with the client code (or at least "stubs" must be provided, which simulate the dependencies through library functions). The consequence is that PDG-based IFC is not compositional: security of a program cannot be inferred just from the security of its components. Only recently, compositional PDGs were tackled (see section 9).

4.3 PDGs and Slicing for Multi-Threaded Programs

For multi-threaded programs operating on shared memory, PDGs are extended by so-called *interference dependencies*[12] which connect an assignment to a variable in one thread with a read of the same variable in another thread [23]. Figure 7 shows a small example with two interference edges. The backward slice from node $\mathtt{x} = \mathtt{y} + \mathtt{1}$ consists of the grey nodes.[13] Now assume `x` and `b` are public; `a` and `y` are secret. The illegal explicit flow from `y` to `x` is captured in the PDG by (among others) the data dependence $\mathtt{y} \overset{D}{\to} \mathtt{x} = \mathtt{y} + \mathtt{1}$; the illegal implicit flow from `a` to `b` is captured by the control dependence $\mathtt{a} > \mathtt{0} \overset{C}{\to} \mathtt{b} = \mathtt{0}$.

```
                                void thread_2():
int x, y;                         a = y;
                                  x = <input>;
void thread_1():                  if a > 0
  x = y + 1;                        b = 0;
  y = 0;                          else
                                    y = 0;
```

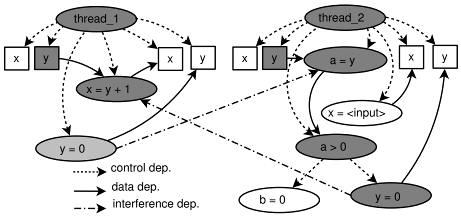


Fig. 7 PDG for two threads with interference dependences.

The simplest slicer for multi-threaded programs is the iterated two-phase slicer (I2P) [33,14]. I2P uses the context-sensitive HRB algorithm[14], but does not traverse interference edges directly. Instead, I2P adds the starting point of interference edges to a work list and thus repeatedly applies the intra-thread HRB backward slice algorithm for every interference edge.

I2P can be improved by using May-happen-in-parallel (MHP) information. Often MHP analysis can prove that, due to locks or the fork/join structure, certain statements cannot happen in parallel. Such information can be used to prune interference dependencies, drastically improving scalability and precision. Various MHP algorithms for Java have been published, e.g., [34,26,8].

I2P is useful in practice, but – even with MHP – does not deliver maximal precision, as it ignores the issue of *time travel.* This phenomenon can be explained as follows. In any real information flow or leak, the information source must be executed before (i.e. physically earlier) than the sink. If for all possible interleavings the source executes after the sink, flow is impossible. This applies in particular to interference dependencies. Naive traversal of interference dependencies however can indicate "flows" where the source is executed after the sink. No scheduler will ever generate such an execution trace.

Figure 7 presents an example: the two dashed interference edges exclude each other, because flow along the first requires `thread_1` to execute before `thread_2`, and flow along the second requires `thread_2` to execute before `thread_1`. Hence the light grey node $\mathtt{y} = \mathtt{0}$ in `thread_1` cannot influence the node $\mathtt{x} = \mathtt{y} + \mathtt{1}$. The example also demonstrates that interference dependences are – in contrast to data and control dependences – not transitive. In fact, this intransitivity is the root cause for time travel. As a consequence, summary edges in HRB can never contain flow through interference edges,

[12] "interference dependencies" have nothing to do with "noninterference" in the IFC sense; the naming is for historical reasons.

[13] where the light grey node is pruned by time-sensitive analysis; see below.

[14] HRB slicing has two phases, hence the name I2P.

as summary edges are transitive. This explains why I2P needs a worklist for interference edges.

A *time-sensitive analysis* discards impossible, "time-travelling" flows.[15] I2P is context-sensitive inside threads, but not time-sensitive. Time-sensitive interprocedural slicing algorithms are very complex, and cannot be described in detail here; for many years, no scalable algorithm existed. Today, our new algorithms, which are based on earlier work by Krinke and Nanda [23,33] allow analysis of at least a few kLOC of full Java [7,9,8]. Benchmarks have shown that I2P precision (i.e. slice size) is ca. 25% worse than the best known time-sensitive slicer [8].

Lock-sensitivity. Most concurrent programs contain explicit locks and synchronization. However, many MHP algorithms ignore locks – they only analyse fork-join structure, perhaps in a context-sensitive manner. The reason is that lock-sensitive MHP needs a precise must-alias analysis, which is notoriously difficult and expensive [8]. Section 9 discusses approaches to lock-sensitive PDGs. Note that a lock-sensitive MHP only improves precision, but does not affect correctness of PDGs and IFC.

5 A slicing-based static LSOD check and the RLSOD criterion

5.1 The static check

Let us now come back to our LSOD criterion and its sound, static approximation through PDGs and slicing. First we emphasize that dynamic dependencies and thus *DFS* and *DCD* can be soundly approximated by static slices and PDGs. In particular, the following soundness properties for *BS* and *FS* hold [18,16]:

Observation. Let o be an operation in trace T.

1. If $DFS_T(o) = \{p_1, \ldots, p_k, \ldots\}$ then
$$\{stmt(p_1), \ldots, stmt(p_k), \ldots\} \subseteq FS(stmt(o))$$
2. If $stmt(o) \notin BS(stmt(p))$, then $p \notin DFS_T(o)$, and it is guaranteed that o cannot influence p through explicit or implicit flow.

Due to these properties, the three conditions of theorem 1 can naturally be checked using slicing for concurrent programs. Furthermore, a sound and precise MHP analysis is needed; we write $MHP(s, s')$ if this analysis concludes that PDG nodes $s, s' \in N$ may be executed in parallel. Remember that $cl(s)$ is the classification (H or L) of PDG node s; we assume a two-element security lattice where $L \sqsubseteq H$. $def(s)$ are the variables defined (written) in s, while $use(s) \supseteq def(s)$ are the variables used (read or written) in s.

Then the following static conditions are sufficient to guarantee the three dynamic conditions in theorem 1:

Static check. For all $s, s', s'' \in N$,

1. $cl(s) = L \wedge cl(s') = H \implies s' \notin BS(s)$
2. $MHP(s, s') \wedge \big(\exists v : v \in def(s) \wedge v \in use(s')\big)$
$\wedge\, cl(s'') = L \implies s, s' \notin BS(s'')$
3. $MHP(s, s') \implies \big(cl(s) = H \vee cl(s') = H\big)$

The first condition is also the basis for the sequential IFC [16], the second disallows data conflicts between low-observable operations, and the last demands that no two low-observable operations are in an order conflict. Thus Zdancewic's original idea, namely to disallow conflicts between publicly observable effects, is naturally implemented through slicing of concurrent programs.

Example 1. For the program in Figure 1 right, we have $cl(\texttt{inputPIN()}) = H$, $cl(\texttt{print(x)}) = cl(\texttt{print(2)}) = L$. There are no explicit or implicit leaks, so rule 1 does not fire. But $MHP(\texttt{print(x)}, \texttt{print(2)})$, so there is an order conflict: $\texttt{print(x)} \overset{oconf}{\leftrightsquigarrow} \texttt{print(2)}$. Indeed rule 3 rejects the program immediately. Furthermore, $MHP(\texttt{print(x)}, \texttt{x = 1})$, thus there is a data race concerning `x`. Since any PDG node is in its own backward slice, rule 2 rejects the example as well, where $s' = s'' = \texttt{print(x)}$.

Example 2. Consider again Figure 3. The programs in the right are accepted, as there are no explicit, implicit or probabilistic flows. Both programs contain termination channels which are however not discovered by the static check. The top left program contains an additional implicit flow ($\texttt{inputPIN()} \in BS(\texttt{print("x")})$), and thus is rejected by the static check – even though its behaviour is identical to Figure 3 bottom right. But the check analyses program text, not program behaviour (and behaviour equivalence is undecidable). For the program bottom left, which again behaves identical to top left and bottom right, the static check discovers the same implicit flow.

Now remember that for this program – in contrast to the top left program – definition 6 is fulfilled: it does not discover the termination leak. The static check is however a sound overapproximation of definition 6: in rare cases, it rejects programs which are accepted by definition 6. In particular, it rejects more termination leaks than definition 6, but it never misses non-termination leaks. In Figure 3, the static check thus discovers 2 of 4 termination leaks, while definition 6 discovers only 1 of 4 termination leaks. The example indicates that the static check is, after all, "only 50% termination insensitive", however this number has not been validated in realistic experiments.

5.2 RLSOD: allowing noncritical conflicts

Often LSOD is criticised as it may prohibit useful nondeterminism (due to rules 2 and 3). For example, if several threads nondeterministically write to the same public file, LSOD as described so far will always prohibit such nondeterministic writes due to rule 3.

[15] "time sensitivity" has nothing to do with "timing leaks" in the IFC sense, the naming is for historical reasons.

```
class Example {
 static volatile int[] buffer = new int[100];
 int y;

 static class A extends Thread {
   public void run() {
     int newentry = y;
     ... // code that enters new entry into buffer
     ... // buffer may contain high values,
     ... // but these are not observable!
   }
 }

 static class B extends Thread {
   public void run() {
     y = inputPIN();
     print("debug␣info");
     ... // code that consumes entries from buffer
     print("more␣debug␣info");
   }   }
}
```

```
class Example {
 static int x;

 static class A extends Thread {
   public void run() {
     x = 0;
     print(x);
   }
 }

 static class B extends Thread {
   public void run() {
     int y = inputLowData();
// low input instead of high PINs
     while (y > -1)
       y--;
     x = 1;
     print(2);
   }   }
}
```

Fig. 8 The RLSOD criterion does not prohibit useful nondeterminism

But note that *a race or conflict is only harmful if there is a possible control flow from a high node to the conflicting nodes* – otherwise the conflict cannot cause a leak. In particular, programs without high sources may contain arbitrary low determinism, and arbitrary high nondeterminism is allowed if it cannot influence low events. This important observation allows to prevent spurious rejection of useful nondeterminism.

The result is a criterion which, strictly speaking, cannot be called LSOD any more, as LSOD definition 6 absolutely disallows any low-nondeterminism. We call the new criterion *RLSOD* (relaxed LSOD). It is easy to implement: potential control flow can be checked in the threaded control flow graph (see section 6). Thus item 3 in the above static check now reads

3'. $\big(MHP(s,s')\wedge\exists a : cl(a) = H\wedge(flow(a,s)\vee flow(a,s'))\big) \implies \big(cl(s) = H \vee cl(s') = H\big)$

Examples. In Figure 2 right, neither `l=0` nor `print(l)` can be reached from high events, as both are initial in their thread. Thus the example is RLSOD even though it is not LSOD (cmp. example 1 in section 3.2). In Figure 8 top, the `inputPIN` cannot influence the writes to the public debug file, hence the nondeterminism present in the high-influenced parts is allowed. In Figure 8 bottom, there is no high input and hence all low conflicts are allowed. Thus both examples are RLSOD. Similar anti-LSOD examples from the literature are RLSOD likewise. Thus the long-standing statement “LSOD prohibits useful nondeterminism” has lost its foundation.

Note however that low-nondeterminism can be insecure – even if the RSLOD criterion (or any PN criterion) is satisfied – if the scheduler can be manipulated. A manipulated scheduler could, for example, read a high value before scheduling (e.g., `h=0` or `h=1`), and then schedule low-determinism (e.g., in `l = 0 || l = 1;`) in a way such that the high value is copied to a low variable. This is the reason why some authors disallow schedulers which read high data [42], while others favor scheduler-specific IFC and PN [20,35] (see also section 8). In the current work, the possibility of manipulated schedulers is ignored, and low nondeterminism is considered secure if *in the program code* it is not influenced by high events. We consider this approach consistent with the notion of “language-based” security.

6 Implementation

We assume that all PDG nodes $n \in N$, as well as input/output streams, are annotated (classified) with a security level $cl(n)$. It is enough to annotate inputs $I \subseteq N$ and outputs $O \subseteq N$ (also called sources and sinks), as the security level for intermediate nodes $n \in N \setminus (I \cup O)$ can be determined by a fix-point iteration similar to data flow analysis on the PDG [16]. The analysis can handle arbitrary lattices of security levels, not just the two-element lattice $L \sqsubseteq H$.

For the implementation, the PDG is enriched with data and order conflict edges. The result is called a CPDG (conflict-enriched program dependency graph).

Definition 8 (Data and order conflict edges) Let $G = (N, \rightarrow)$ be a PDG. Let $m, n \in N$ where $MHP(m, n)$. There is a data conflict edge $m \rightarrow_{dconf} n$ to G if m defines a variable v that is used or defined by n: $\exists v : v \in def(m) \wedge v \in use(n)$. There is an order conflict edge $m \leftrightarrow_{oconf} n$ to G if both nodes are classified as sources or sinks: $m, n \in I \cup O$.

Example. Figure 9 shows the CPDG of the program on the right hand side of Figure 1. The example assumes that `y = inputPIN()` is classified as a source of high data and `print(x)` and `print(2)` are classified as sinks of low data. The CPDG contains two order conflict edges, one between `print(x)` and `print(2)` and one between `print(x)` and `y = inputPIN()`, and three data conflict edges, from `x = 0` to `x = 1`, from `x = 1` to `x = 0` and from `x = 1` to `print(x)`.

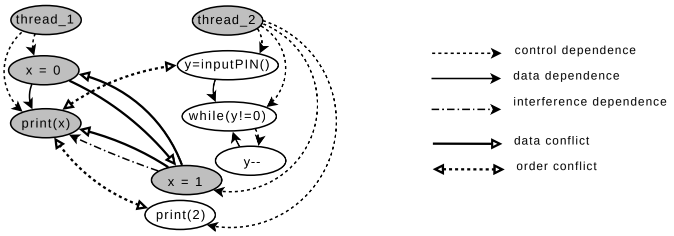


Fig. 9 PDG of the program on the right side of Fig. 1, enriched with data and order conflict edges. The grey nodes denote the slice for node `print(x)`. Note that the slice ignores conflict edges.

Definition 9 (TCFG) A Threaded Control Flow Graph (TCFG) consists of the interprocedural CFGs for the individual threads, connected by fork and join edges.

A formal definition of TCFGs can be found in [8]. A path in the TCFG is written $a \rightarrow^*_{TCFG} b$ (or $\mathit{flow}(a, b)$ for short[16]). Once CPDG and TCFG have been constructed, the IFC checker proceeds as follows:

1. Compute $BS(s)$ for every sink $s \in O$.
2. If the backward traversal encounters a source $i \in I$ where $cl(i) \not\sqsubseteq cl(s)$, then the program may leak data of level $cl(i)$ via explicit or implicit flow and is rejected. This criterion is also used in our sequential IFC [16].
3. If the traversal encounters an incoming data conflict $m \rightarrow_{dconf} n$, the program may contain a probabilistic data channel and is rejected.
4. If the traversal encounters an order conflict $m \leftrightarrow_{oconf} n$, check if the order conflict is low-observable, i.e. $cl(m) = cl(n) = L$. If so, the program may contain a probabilistic order channel and is rejected.

Example. Consider Figure 9. The backward slice for `print(2)` encounters the order conflict edge between `print(2)` and `print(x)`, so the program may contain a probabilistic order channel. The slice for `print(x)`, highlighted grey in Figure 9, encounters all data conflict edges, so the program may contain a probabilistic data channel as well, whereas its implicit and explicit flow is secure.

In order to allow noncritical conflicts (RLSOD criterion, see section 5.2), we change item 4 in the algorithm as follows:

4'. If the traversal encounters an order conflict $m \leftrightarrow_{oconf} n$, check if $cl(m) = cl(n) = L$. If so, check in the TCFG if any node that can be executed before both conflicting nodes is a high source: $\exists\ a \in N : cl(a) = H \wedge a \rightarrow^*_{TCFG} m \wedge a \rightarrow^*_{TCFG} n$. If so, the program may contain a probabilistic order channel and is rejected. If not, the conflict is not critical.

In the implementation, rules 2. - 4. are integrated into a backward I2P slicer. A time-sensitive slicer would be more precise, but would make the algorithm much more complex and expensive. In practice, I2P precision is often sufficient; time-sensitive slicing can have exponential runtime and thus should only be applied if the I2P approach produces false alarms.

Algorithm 1 Information flow control for concurrent programs.

```
Input: A classified CPDG $G = (N, E)$, its TCFG $C$, a security lattice $\mathfrak{L}$.
Output: 'true' if the program is LSOD (up to declassification and harmless conflicts), 'false' otherwise.
Let src($n$) be the source level of node $n$ (= $\bot$ if $n$ is not a source)
Let sink($n$) be the the sink level of node $n$ (= $\top$ if $n$ is not a sink)

/* Check implicit and explicit flow: */
Let flow($G, C, \mathfrak{L}$) be a function that returns false if $G$ contains illicit implicit or explicit flow.
if flow($G, C, \mathfrak{L}$) == false then
  return false
/* Scan the program for probabilistic channels. */
/* Check sources: */
for all $n \in N$ : src($n$) $\neq \bot$ do
  if prob($G, C, n$, src($n$), $\mathfrak{L}$) == false then
    return false
/* Check sinks: */
for all $n \in N$ : sink($n$) $\neq \top$ do
  if prob($G, C, n$, sink($n$), $\mathfrak{L}$) == false then
    return false
return true
```

Algorithms 1, 2 and 3 present detailed pseudocode. Algorithm 1 receives a CPDG in which sources and sinks are already classified and which already contains the order conflict edges, the corresponding TCFG and the security lattice in charge. It then runs a slicing-based check of the implicit and explicit flow (that is, it checks rule 2; in fact the algorithm from [16] is used). If the program passes that check, it is scanned for probabilistic channels by checking rules 3 and 4. This is done by Algorithm 2.

Algorithm 2 receives the CPDG, the TCFG, the security lattice and a source or sink s of a certain security level l. The algorithm first checks whether s is involved in a low-observable order conflict that can be preceded by a source of high data. This task is delegated to the auxiliary procedure `benign` in Alg. 3. After that, it executes an extended I2P slicer which additionally checks if s is potentially influenced by a data conflict whose nodes can be preceded by a source of high data. This check is again delegated to procedure `benign`. The "phase 1" and "phase 2" in the I2P loop are just the two phases of the HRB slicer, which is inlined into the I2P algorithm.

Procedure `benign` implements rule 4. First it checks whether the given conflict is an order conflict and whether it is low-observable, which it is if both conflicting nodes are visible to the attacker. To allow noncritical conflicts,

[16] The latter notation was already used in section 5.2

Algorithm 2 Procedure `prob` detects probabilistic channels.

```
Input: An CPDG $G = (V, E)$, its TCFG $C$, a node $s$, its security level $l$, the security lattice $\mathfrak{L}$.
Output: 'false' if $s$ leaks information through a probabilistic channel, 'true' otherwise.
/* Check G for probabilistic order channels. */
/* inspect order conflicts: */
for all $m \leftrightarrow_{oconf} s$ do
  if benign$(C, m, n, oconf, \mathfrak{L}, x)$ == false then
    return false
/* Check G for probabilistic data channels. */
/* initialize the modified I2P-slicer*/
$W = \{s\}$ /* a worklist */
$M = \{s \mapsto true\}$ /* maps visited nodes to true (phase 1) or false (phase 2) */
repeat
  remove first node $n$ from $W$
  /* look for data conflicts */
  for all $m \rightarrow_{dconf} n$ do
    if benign$(C, m, n, dconf, \mathfrak{L}, l)$ == false then
      /* conflict is harmful */
      return false
  /* proceed with standard I2P slicing */
  for all $e = m \rightarrow n$ where $e$ is not a conflict edge do
    /* if m hasn't been visited yet or we are in phase 1 and m has been visited in phase 2 */
    if $m \notin$ dom $M \vee (\neg M(m) \wedge (M(n) \vee e$ is a concurrency edge )) then
      /* concurrency edges comprise interference edges, fork-in, fork-out and join-out edges */

      if $M(n)$ /* we are in phase 1 */ $\vee e$ is not a parameter-in or call edge then
        append $m$ to worklist $W$
      /* determine how to mark m: */
      if $M(n) \wedge e$ is a parameter-out edge then
        /* we are in phase 1 and e is a parameter-out edge: mark m with phase 2 */
        $M = M \cup \{m \mapsto false\}$
      else if $\neg M(n) \wedge e$ is a concurrency edge then
        /* we are in phase 2 and e is a concurrency edge: mark m with phase 1 */
        $M = M \cup \{m \mapsto true\}$
      else
        /* mark m with the same phase as n */
        $M = M \cup \{m \mapsto M(n)\}$
until $W = \emptyset$
return true /* no probabilistic channels */
```

Algorithm 3 Procedure `benign` identifies benign conflicts.

```
Input: A TCFG $C = (N, E)$, two conflicting nodes $a$ and $b$, the kind $e$ of the conflict, a security lattice $\mathfrak{L}$, a security level $l \in \mathfrak{L}$.
Output: 'true' if the conflict is harmless, 'false' otherwise.
Let reaches$(m, n, C)$ return 'true' if there exists a realizable path from node $m$ to node $n$ in $C$.

/* Check visibility of order conflicts. */
if $e == oconf$ then
  $x = ($src$(a) \neq \bot \wedge$ src$(a) \sqsubseteq l) \vee ($sink$(a) \neq \top \wedge$ sink$(a) \sqsubseteq l)$ /* is 'a' visible? */
  $y = ($src$(b) \neq \bot \wedge$ src$(b) \sqsubseteq l) \vee ($sink$(b) \neq \top \wedge$ sink$(b) \sqsubseteq l)$ /* is 'b' visible? */
  if $\neg x \vee \neg y$ then
    return true /* the order conflict is not visible */
/* Check if a source of high data may execute before the conflicting nodes. */
for all $n \in N$ do
  if src$(n) \not\sqsubseteq l$ then
    if (reaches$(n, a) \vee MHP(n, a)) \wedge ($reaches$(n, b) \vee MHP(n, b))$ then
      return false /* the conflict is harmful */
return true /* the outcome of the conflict cannot be influenced by high data */
```

it additionally checks if the conflicting nodes are preceded by a high source n. This is the case if n reaches them on realisable paths in the TCFG or if it may happen in parallel to them.

Example 1. The sequential programs in Figure 3 right are accepted by algorithm 1, as there are no explicit, implicit or probabilistic flows. Both programs contain termination channels which, however, are not discovered by algorithm 1. On the other hand, the left programs contain additional implicit flow and thus are rejected by algorithm 1 – even though their behaviour is identical to the bottom right program. This seeming inconsistency is a deliberate consequence of allowing termination leaks, as discussed at length in section 2.

Example 2. Consider Figure 9. Algorithm 1 passes the `flow` call successfully (no implicit or explicit flows, as checked by sequential IFC for every thread). It then calls algorithm 2 for the high source `y = inputPIN()` and for the two low sinks `print(x)` and `print(2)`. Tracing the last call, algorithm 2 sees the order conflict between `print(2)` and `print(x)`, hence it calls algorithm 3. The latter discovers visibility of the conflict in the second main phrase of the first **if**, which prevents the conflict to be benign – algorithm 2 immediately returns `false`. If we nevertheless trace the algorithm a little further, the worklist for the I2P slicer is initialized with `print(x)`. In the first iteration, the **for** loop discovers m =`x=0;` and m =`x=1;` to be in immediate data conflict with n =`print(x)`. Algorithm 3 discovers that a source of high data, namely `y=inputPIN();` can reach `print(x)`; hence the data conflict is harmful. The example shows that the RLSOD check is terminated as soon as a leak is found; it can also be modified to return a list of *all* leaks, where a PDG path is given for every leak.

7 Evaluation

The above algorithms have been implemented for full Java, and integrated into JOANA. To our knowledge, no other evaluations of LSOD or PN precision or scalability have been published, hence we cannot compare our implementation to other algorithms.

7.1 Precision

To assess precision, we analysed examples from the literature. Our first example is from [44] (Figure 10); it was explained in section 3. Using JOANA, we classified `PIN = Integer.parseInt(args[0])` as a high source and `System.out.println(result)` as a low sink. No other classifications were necessary. Algorithm 1 detected an order conflict which depends on high data: the assignments to `result` in threads `Alpha` and `Beta` are conflicting, and the outcome of the conflict is influenced by the values of `trigger0` and `trigger1`, which in turn are changed dependent on `PIN`'s value in thread `Gamma`. Thus, this program contains a probabilistic data channel which leaks information about `PIN` to `result` and is rejected by RLSOD.

Our second example in Figure 11 is from [32]. The program is probabilistic noninterferent, which is however difficult to discover. The program manages a stock portfolio of Euro Stoxx 50 entries. The portfolio data, `pfNames` and `pfNums`, is secret, hence neither the Euro Stoxx request by `EuroStoxx50`, nor the final message sent to a commercials provider may contain any information about the portfolio. Indeed `Portfolio` and `EuroStoxx50` do not interfere, thus the Euro Stoxx request does not leak information about the portfolio. The message sent to the commercials provider is not influenced by the values of the portfolio, either, because there is no explicit or implicit flow from the secret portfolio values to the sent message. Furthermore, the two outputs have a fixed relative ordering, as `EuroStoxx50` is joined before `Output` is started. Hence, the program should be considered secure.

We classified the two statements reading the portfolio from storage, `pfNames = getPFNames()` and `pfNums = getPFNums()`, as high sources and the output flushes `nwOutBuf` in `EuroStoxx50` and at the end of `main` as low sinks; other classifications were not necessary. The challenge of this program is to detect that `EuroStoxx50` is joined before `nwOutBuf` is flushed in the `main` procedure, because otherwise it cannot be determined that the two flushes of `nwOutBuf` have a fixed execution order. And then the program would have to be rejected because the resulting order conflict is influenced by both sources.

```
class Alpha extends Thread {
  public void run() {
    while (mask != 0) {
      while (trigger0 == 0) ; /* busy wait */
      result = result | mask;
      trigger0 = 0;
      maintrigger++;
      if (maintrigger == 1) trigger1 = 1;
    }
  }
}
class Beta extends Thread {
  public void run() {
    while (mask != 0) {
      while (trigger1 == 0) ; /* busy wait */
      result = result & ~mask;
      trigger1 = 0;
      maintrigger++;
      if (maintrigger == 1) trigger0 = 1;
    }
  }
}
class Gamma extends Thread {
  public void run() {
    while (mask != 0) {
      maintrigger = 0;
      if ((PIN & mask) == 0) trigger0 = 1;
      else trigger1 = 1;
      while (maintrigger < 2) ; /* busy wait */
      mask = mask / 2;
    }
  }
}

class SmithVolpano {
  static int maintrigger, trigger0,
             trigger1 = 0, PIN, result = 0;
  static int mask = 2048; // a power of 2

  public static void main(String[] args)
                     throws Exception {
    PIN = Integer.parseInt(args[0]);
    Thread a=new Alpha();
    Thread b=new Beta();
    Thread g=new Gamma();
    g.start(); a.start(); b.start();
// start all threads
    g.join(); a.join(); b.join();
// join all threads
    System.out.println(result);
  }
}
```

Fig. 10 Example from Smith and Volpano [44]

Our MHP analysis, together with context-sensitive points-to analysis, was able to detect that the joins of the threads are must-joins, which enabled the RLSOD algorithm to identify that there is no order conflict between the two flushes of `nwOutBuf` (cmp. definitions 7, 8), which in turn avoided false alarms in the “benign” check. Therefore no probabilistic channel was reported. To our knowledge, algorithm 1 is the first implementation of LSOD or PN, which was precise enough to verify that this example is noninterferent.

More case studies can be found in [8].

```java
class Mantel {
  // to allow mutual access, threads are global variables
  static Portfolio p = new Portfolio();
  static EuroStoxx50 e = new EuroStoxx50();
  static Statistics s = new Statistics();
  static Output o = new Output();

  static BufferedWriter nwOutBuf =
      new BufferedWriter(new
            OutputStreamWriter(System.out));
  static BufferedReader nwInBuf =
      new BufferedReader(new
            InputStreamReader(System.in));
  static String[] output = new String[50];

  public static void main(String[] args)
         throws Exception {
    // get portfolio and eurostoxx50
    p.start(); e.start();
    p.join();  e.join();
    // compute statistics and generate output
    s.start(); o.start();
    s.join();  o.join();
    // display output
    stTabPrint("No.\t␣|␣Name\t␣|␣Price\t␣|␣Profit");
    for (int n = 0; n < 50; n++)
      stTabPrint(output[n]);
    // show commercials
    stTabPrint(e.coShort+
              "Press␣#␣to␣get␣more␣information");
    char key = (char)System.in.read();
    if (key == '#') {
      System.out.println(e.coFull);
      nwOutBuf.append("shownComm:"+e.coOld);
      nwOutBuf.flush();
// public output
    }
  }
}

class Portfolio extends Thread {
  int[] esOldPrices, pfNums;
  String[] pfNames; String pfTabPrint;

  public void run() {
    pfNames = getPFNames();
// secret input
    pfNums = getPFNums();
// secret input
    for (int i = 0; i < pfNames.length; i++)
      pfTabPrint += pfNames[i] + "|" + pfNums[i];
  }

  int locPF(String name) {
    for (int i = 0; i < pfNames.length; i++)
      if (pfNames[i].equals(name)) return i;
    return -1;
  }
}
```

```java
class EuroStoxx50 extends Thread {
  String[] esName = new String[50];
  int[] esPrice = new int[50];
  String coShort;
  String coFull;
  String coOld;

  public void run() {
    try {
      nwOutBuf.append("getES50");
      nwOutBuf.flush();
// public output
      String nwIn = nwInBuf.readLine();
      String[] strArr = nwIn.split(":");
      for (int j = 0; j < 50; j++) {
        esName[j] = strArr[2*j];
        esPrice[j] = Integer.parseInt(strArr[2*j+1]);
      }
      // commercials
      coShort = strArr[100];
      coFull = strArr[101];
      coOld = strArr[102];
    } catch (IOException ex) {}
  }
}

class Statistics extends Thread {
  int[] st = new int[50];
  int k = 0;

  public void run() {
    k = 0;
    while (k < 50) {
      int ipf = p.locPF(e.esName[k]);
      if (ipf > 0)
        st[k] = (p.esOldPrices[k] -
                 e.esPrice[k]) * p.pfNums[ipf];
      else
        st[k] = 0;
      k++;
    }
  }
}

class Output extends Thread {
  public void run() {
    for (int m = 0; m < 50; m++) {
      while (s.k <= m) ;  /* busy wait */
      output[m] = m+"|"+e.esName[m]+"|" +
                  e.esPrice[m]+"|"+s.st[m];
    }
  }
}
```

Fig. 11 Example from Mantel et al [32], converted to Java. For brevity, some methods are not shown.

7.2 Runtime Behaviour

We applied our algorithm to a benchmark of 8 programs between 200 and 3000 LOC (taken from the Bandera benchmark and the JavaGrande benchmark). We used three different security lattices, called A, B, C; with 3, resp. 22, resp. 254 elements. For each program and lattice, we randomly chose 10 sources and 10 sinks, then 33 sources and 33 sinks, and finally 100 sources and 100 sinks of random security levels, and analysed the classified programs. We measured the total execution times, and separate execution times of the scan for probabilistic channels and of the scan for explicit or implicit flow. Every test was run ten times.

Table 1 shows the average execution times (measured on a rather old standard PC). It contains one row for each combination of program and lattice, i.e. row 'LG + A' contains the results for program "LaplaceGrid" and lattice A. The numbers reveal that the most important factor influencing the runtime behaviour is, besides the sheer size of the program, the number of sources and sinks. With an increasing number of sources and sinks

Table 1 Average execution times of algorithm 1 for different programs, lattices and numbers of sources and sinks (in seconds).

Name +	sources x sinks		
Lattice	10 x 10	33 x 33	100x 100
LG + A	1.6	4.8	21.2
LG + B	1.6	5.8	29.7
LG + C	2.0	9.5	170.7
(200 LOC)			
SQ + A	5.9	17.2	54.0
SQ + B	5.5	17.3	68.0
SQ + C	5.8	21.1	162.5
(350 LOC)			
KK + A	25.7	58.5	170.0
KK + B	22.1	57.5	187.8
KK + C	25.2	64.9	256.2
(600 LOC)			
RT + A	8.9	25.3	99.3
RT + B	7.3	23.9	116.1
RT + C	8.4	27.2	175.1
(950 LOC)			
MC + A	17.1	53.3	224.2
MC + B	18.7	53.3	173.6
MC + C	17.5	54.8	205.0
(1400 LOC)			
JS + A	2.2	5.2	18.8
JS + B	2.4	5.6	20.3
JS + C	2.4	5.8	40.7
(500 LOC)			
PO + A	6.4	18.1	54.6
PO + B	7.4	19.1	66.8
PO + C	7.0	20.4	89.8
(2000 LOC)			
CS + A	19.4	52.5	153.3
CS + B	21.5	52.1	160.2
CS + C	21.0	53.6	177.3
(3000 LOC)			

Table 2 The percentage share of the probabilistic channel detection among the overall execution times.

Name +	sources x sinks		
Lattice	10 x 10	33 x 33	100x 100
LG + A	53	48	62
LG + B	56	54	73
LG + C	58	73	94
SQ + A	44	39	44
SQ + B	43	40	55
SQ + C	43	50	80
KK + A	57	45	43
KK + B	58	46	45
KK + C	58	48	58
RT + A	44	40	44
RT + B	49	41	50
RT + C	47	46	67
MC + A	45	38	38
MC + B	43	38	39
MC + C	44	40	47
JS + A	52	46	41
JS + B	46	46	46
JS + C	53	54	71
PO + A	46	38	35
PO + B	45	37	36
PO + C	43	39	46
CS + A	38	29	28
CS + B	34	30	28
CS + C	32	31	34

the size of lattice C eventually became the dominating cost factor. A more detailed analysis reveals that the sequential IFC check in algorithm 1 needs about the same execution time as the additional probabilistic checks.

Table 2 shows the percentage share of the probabilistic channel detection among the overall execution times. The remaining time was consumed by the algorithm of Hammer et al. [16], which is employed for verifying the explicit and implicit flow. The results show that the two checks were similarly fast. However, the probabilistic check seems to decline in performance for large lattices, compared to Hammer et al.'s algorithm; which is consistent with the above-mentioned performance sensitivity for very large lattices.

8 Discussion and Related Work

In the following, we compare our definition of LSOD with PN and LSOD definitions from the literature.

8.1 Weak Probabilistic Noninterference

```
void thread_1():
  h = inputPIN();
  if (h < 0)
    h = h * (-1);
  l = 0;

void thread_2():
  x = 1;
```

```
void thread_1():
  h = inputPIN();
  if (h == 0)
    h = h + 2;
  else
    h = h - 2;
  l = 0;

void thread_2():
  l = 1;
```

Fig. 12 Two examples comparing LSOD and WPN. We assume that Smith and Volpano's technique classifies variables `h` and `x` as high and `l` as low, and that our technique classifies `h = inputPIN()` as a high input and `l = 0` and `l = 1` as low output. The left program is accepted by our condition and rejected by theirs, the right program is rejected by ours and accepted by theirs.

Smith and Volpano's *weak probabilistic noninterference* (WPN) property [43,48] enforces probabilistic noninterference via *weak probabilistic bisimulation.* A program is WPN if for each pair of low-equivalent inputs, each sequence of low-observable events caused by one input can be caused by the other input with the same probability.

WPN addresses explicit and implicit flow and probabilistic channels. Like in our analysis, timing channels and termination channels are excluded (which permits the probabilistic bisimulation to be weak). This renders their interpretation of low-observable behaviour very similar to ours: It consists of a sequence of low-observable events, but lacks information about the time at which such an event occurs. The major difference concerning low-equivalent behaviour is that their definition disallows low-observable events to be delayed infinitely in one low-observable behaviour and to be executed in the other. Thus, WPN is stricter with respect to termination channels and only permits the sheer termination of the program to differ.

WPN globally partitions the program variables into high and low, and the attacker is able to see all low variables at any time. In contrast, our attacker can only see low operations once they are executed. In particular, in our flow-sensitive approach the same variable can be low at one program point or high at another, dependent on the context.

Thus the WPN attacker is generally more powerful than ours, because we assume that only low operations/events and their low operands are visible to the attacker. The price to be paid is flow insensitivity, resulting in strange false alarms, as we saw in e.g., Figure 2. It depends on the application context which attacker model is more realistic.

Smith and Volpano's security-type system lacks a detection of conflicts. Probabilistic channels are prevented by forbidding assignments to low variables sequentially behind conditional structures, which is very restrictive. The program on the left side of Fig. 12 is rejected by WPN, because the running time of the `if`-structure depends on high data and is followed sequentially by `l = 0`. However, it does not contain a probabilistic channel because `l = 0` is not involved in an order conflict or influenced by a data conflict. It therefore satisfies RLSOD.

WPN assumes that a single statement has a fixed running time. Thus programs like in Figure 12 (right) are accepted by WPN, because the branches of the `if`-structure have equal length and thus different values of `h` do not alter the probabilities of the possible ways of interleaving of `l = 0` and `l = 1`. We explicitly aim to reject such programs, arguing that different running times of `h = inputPIN()` could already cause a probabilistic channel, and our security constraint rejects the program because of the data conflict between `l = 0` and `l = 1`.

Smith and Volpano's security-type system is restricted to probabilistic schedulers and breaks, for example, in the presence of a round-robin scheduler [43]. (R)LSOD holds for every scheduler.

8.2 Strong Security

Sabelfeld and Sands' security property *strong security* [42] addresses implicit and explicit flow, probabilistic channels and termination channels. It enforces probabilistic noninterference for all schedulers whose decisions are not influenced by high data. It makes the following requirements to a program p and all possible pairs (t, u) of low-equivalent inputs: Let $\mathfrak{T}$ and $\mathfrak{U}$ be the set of possible program runs resulting from t and u. For every $T \in \mathfrak{T}$, there must exist a low-equivalent program run $U \in \mathfrak{U}$.

Even though it looks like a possibilistic property, strong security is capable of preventing probabilistic channels, the trick being the definition of low-equivalent program runs: Two program runs are low-equivalent if they have the same number of threads and they produce the same low-observable events and create or kill the same number of threads at each step under any scheduler whose decisions are not influenced by high data. This 'lock-step execution' requirement allows ignoring the concrete scheduling strategy.

The attacker sees all low variables at all times and is aware of program termination. The values of the low variables, their changes over time and the termination behavior constitute the low-observable behaviour. Strong security assumes that the attacker is not able to see which statement is responsible for a low-observable event, and is designed to identify whether two syntactically different subprograms have equivalent low-observable effects. This makes it possible to identify programs like Figure 13 (left) as secure. Even though the assignments to the low variable `l` are influenced by high data via implicit flow, strong security states that the low-observable behaviour is not, because both branches lead to 0 being assigned to `l`. Our algorithm is not able to recognize this program as secure, the same holds for WPN.

Smith/Volpano and we assume that the attacker cannot exploit termination channels and is able to identify statements responsible for low-observable events, which is seen contrarily by Sabelfeld and Sands.

The requirement of lock-step execution implies that strongly secure programs can be combined sequentially or in parallel to a new strongly secure program (compositionality). Sabelfeld [41] has proven that strong security is the least restrictive security property that provides this degree of compositionality and scheduler-independence. Its compositionality is its outstanding property and an advantage over our LSOD. However, lockstep execution imposes serious restrictions to programs.

Furthermore, the restriction to schedulers which do not touch high data means that any information possibly used by the scheduler, for example the mere number of existing threads, must be classified as low. This in turn means that the classification of a program becomes scheduler-dependent, so the scheduler-independence of strong security is bought by making the classification scheduler-dependent. This allows breaking strong security, by running the program under a scheduler for which the attacker knows the classification of the program to be inappropriate ([42], sec. 4.1).

8.3 LSOD by Zdancewic and Myers

Inspired by Roscoe's earlier work [39], Zdancewic and Myers [53] pointed out that conflicts are a necessary condition of probabilistic channels. They suggested combining a security-type system for implicit and explicit flow with a conflict analysis, arguing that programs without conflicts have no probabilistic channels.

The authors exclude termination channels and probabilistic order channels and justify that by confining the attacker to be a program itself (e.g., a thread). Such an attacker is not able to observe the relative order of low-observable events, because such an observation requires a

```
void main():
  h = inputPIN();
  if (h < 0)
    l = 0;
  else
    l = 0;
```

```
void thread_1():
  h = inputPIN();
  if (h < 0)
    h = h * (-1);
  else
    skip;
  l = 0;

void thread_2():
  x = 1;
```

Fig. 13 Two examples demonstrating the capabilities of strong security. We assume that `h` and `x` are classified as high and `l` as low. The left program is strongly secure, because both branches assign the same value to `l`. The right program is a transformation of the program on the left of Fig. 12, where the additional `skip` statement removes the probabilistic data channel.

probabilistic data channel in which the differing relative orders manifest.

The authors apply the approach to languages with message passing and shared memory. The language provides *linear* communication channels that are used for transmitting exactly one message and thus guarantee conflict-free communication. They present a security-type system for a concurrent calculus λ^{PAR}_{SEC}, that verifies confidentiality of implicit and explicit flow, and verifies that linear channels are used exactly once. The type system guarantees that well-typed programs are LSOD if they are additionally free of data conflicts. Later, Terauchi provided an improved type system for LSOD [47] which does not require confluence of the checked program.

It is interesting to compare their notion of low equivalent traces with ours. Technically, they use a weaker notion of low-equivalence. They do not demand that two traces are low equivalent as in our definition 5 (finite case); they only demand that the projections of the traces onto all individual variables ("location traces") are low equivalent. This implies that low variables undergo the same changes in low-equivalent traces, but not necessarily at the same time. That is, more traces are low equivalent than in our definition – which should also reduce false alarms. Unfortunately, this approach is not flow sensitive. Consider again Figure 2 right: `l` is globally classified low, thus any location trace for `l` has the form $\langle l = 0, l = 0, l = 0, l = h\rangle$. As `h` depends on `inputPIN`, these location traces are not low equivalent for different high inputs, causing a false alarm. In general, the "projection trick" and global variable classification may abstract away from statement order, implying flow insensitivity. In our analysis, `l=h` in Figure 2 is *not* low observable (see discussion in section 3.2), hence the example is RLSOD even though our definition 5 demands more than Zdancewic's.

Zdancewic and Myers' attacker is weaker than ours, as probabilistic order channels are excluded. It is explicitly designed to tackle malicious threads spying out confidential information. Their requirement that programs are completely free of data conflicts (which is much stricter than ours) in practice prevents an application to languages with shared memory, because many programs contain data conflicts, but many such conflicts do not influence low-observable behaviour. We have provided examples of such programs in Figure 8.

8.4 LSOD by Huisman et al.

Huisman et al. [21] pointed out that Zdancewic and Myers' method contains a leak, because its definition of low-equivalent program runs is restricted to the length of the shorter run. They close that leak by strengthening the definition of low-equivalent program runs: assignments to low variables sequentially behind loops iterating over high data are forbidden.

Closing termination channels additionally requires that either both program runs terminate or none of them. They also discover probabilistic order channels, by extending location traces to the set L of low variables: In that case two program runs T and U are low-equivalent if the set of low variables in T and U undergoes the same sequence of changes in both runs up to stuttering.

The authors formalized their different security properties via temporal logics, for which the model-checking problem is decidable if the program in question can be expressed by a finite state machine. This permits a very precise detection of relevant data conflicts, such that total freedom of data conflicts is not required. Hence, their approach can be applied to languages with shared-memory communication.

Closing termination channels has the effect that Huisman's approach is more restrictive than ours. It forbids low-observable events sequentially behind loops iterating over high data. Furthermore, the optional treatment of probabilistic order channels imposes severe restrictions on the analysed programs. As in WPN, each assignment to a low variable is regarded as a low-observable event. Huisman thus requires that two low-equivalent program runs must make the same sequence of changes to low variables. Even if two threads work on completely unrelated low variables, the assignments to these variables must have a fixed interleaving order.

In more recent work, Huisman et al. [20,35] introduced *δ-specific observational determinism.* Since the then-known LSOD criteria all turned out to be either too restrictive or unsound, [20] explicitly reintroduces the scheduling policy δ into the definition, and demands low-equivalency of traces only for traces under the same scheduling policy. Thereby, some restrictions on the original approach can be relaxed, while the model checking approach can be maintained. δ-specific observational determinism is explicitly scheduler dependent; like WPN it is flow insensitive and assumes that variables are globally classified.

Thus Huisman et al. depart again from scheduler independence, because they consider it to be too problematic. In contrast, we consider flow insensitivity to be the troublemaker, as our flow-sensitive LSOD is scheduler independent, sound, precise, and has demonstrated practical applicability. It remains to be seen how the discussion about PN and LSOD will eventually evolve.

8.5 Low-Distinguishability by Mantel, Sands, and Sudbrock

In a recent approach to compositional noninterference for concurrent programs, Mantel, Sands and Sudbrock [29] employ a flow-sensitive security type system [22] and MHP to establish SIFUM-security. Threads are dynamically annotated with "modes", i.e. assumptions and guarantees about variable read and write behaviour in program threads. SIFUM then defines low-distinguishability (which is similar to noninterference) via strong low bisimulation modulo modes. A soundness proof is provided.

Mantel, Sands and Sudbrock certainly improve earlier definitions by exploiting mode information for variables, and by being flow-sensitive. Furthermore, compositionality is an asset not yet provided by PDGs. However [29] is restricted to a fixed number of threads, and does not consider context-, object-, or time-sensitivity; evaluations of an implementation have not yet been reported.

Recently, Sudbrock has shown that for *intra*procedural IFC, PDGs and flow-sensitive type systems such as [22] have exactly the same precision [31]. But note that all known type systems for *inter*procedural IFC are – in contrast to PDGs – not context- or object-sensitive.

9 Future Work

Let us briefly mention some topics in ongoing and future work.

Time sensitivity. Recently, an optional time-sensitive slicer was integrated. We also added an experimental autotuning facility, which automatically switches between variants of pointer analysis (flow-sensitive, context-sensitive) and slicer (time-sensitive and I2P variants). Autotuning is applied to a benchmark of secure programs, until a setup without false alarms has been found. However, more algorithm engineering is needed to balance precision against scalability.

Lock sensitivity. As described in section 2, MHP analysis is crucial for PDG precision and hence for overall IFC precision. The current MHP analysis however does not analyse explicit locks in the program. The latter property is called *lock sensitivity* and has been explored in [5, 6] in the scope of Dynamic Pushdown Networks. We recently integrated this analysis. Preliminary experiments indicate that MHP indeed becomes more precise, as more interference edges are pruned [13].

Declassification. We currently do not provide a declassification mechanism for probabilistic channels, only for sequential channels (see [16] for details). Instead of declassifying probabilistic channels, we consider Zdancewic and Myers' idea of using *linear channels* for deterministic communication between threads [53] more promising. Linear channels can be integrated in form of a library into languages with shared memory. We have recently added a preliminary implementation of such a library.

Compositionality. As explained in section 4.2, the PDG of a complete system cannot be obtained by just combining PDGs of subsystems. We recently investigated mechanisms to overcome this drawback: subsystems can be analysed in isolation, and later plugged into larger systems. However, plug-in of local PDGs is a nontrivial operation, which may require secondary fix-point iteration until global dependencies stabilize.

Machine-checked proofs. It is our long-term goal to formalize our RLSOD check in Isabelle and provide a machine-checked proof for Theorem 1; just as we have provided machine-checked soundness proofs for the sequential (interprocedural) PDG-based IFC [51,50].

Evaluation and comparison. JOANA's RLSOD implementation needs to be applied in larger case studies, and will be compared to other IFC tools.

10 Conclusion

We presented a new method for information flow control in concurrent programs. The method guarantees probabilistic noninterference, and is based on a new variant – named RLSOD – of low-security observational determinism. It turns out that RLSOD can be naturally implemented through slicing algorithms for concurrent programs, which are flow-sensitive, object-sensitive, context-sensitive, and time-sensitive. We also demonstrated how RLSOD fixes some weaknesses of earlier LSOD definitions. In particular, secure low-nondeterminism is allowed by RLSOD. In essence, we demonstrated that flow sensitivity is the key to maintain soundness and precision in IFC for multi-threaded programs.

Our implementation can handle full Java with an arbitrary number of threads. It was applied to examples from the literature and a small benchmark; preliminary experience indicates high precision and scalability for medium-sized programs. Future work will explore declassification for probabilistic channels; we also aim at a machine-checked version of the soundness proof.

Our current work is part of a long-standing project which exploits modern program analysis for software security. The current work demonstrates that IFC analysis of concurrent programs can indeed be improved by applying PDGs and MHP analysis; resulting in algorithms with considerably enhanced precision and scalability.

Acknowledgements We thank the reviewers for their very insightful observations and suggestions. Joachim Breitner and Martin Hecker provided valuable comments.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM.
2. Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. ESORICS*, volume 5283 of *LNCS*, pages 333–348, 2008.
3. David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.
4. David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.
5. Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Concurrency Theory (CONCUR 2005)*, pages 473–487. Springer Verlag, LNCS 3653, 2005.
6. Thomas Martin Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner. Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In *VMCAI*, pages 199–213, 2011.
7. Dennis Giffhorn. Advanced chopping of sequential and concurrent programs. *Software Quality Journal*, 19(2):239–294, 2011.
8. Dennis Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, May 2012. http://pp.info.uni-karlsruhe.de/uploads/publikationen/giffhorn12thesis.pdf.
9. Dennis Giffhorn and Christian Hammer. Precise slicing of concurrent programs – an evaluation of precise slicing algorithms for concurrent programs. *Journal of Automated Software Engineering*, 16(2):197–234, June 2009.
10. Dennis Giffhorn and Gregor Snelting. Probabilistic noninterference based on program dependence graphs. *Karlsruhe Reports in Informatics*, 6, April 2012. http://pp.info.uni-karlsruhe.de/uploads/publikationen/giffhorn12kri.pdf.
11. Jürgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *Proc. 9th SCAM*, pages 105–114, September 2010.
12. Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs - a practical guide. In *Proc. 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2013.
13. Jürgen Graf, Martin Hecker, Martin Mohr, and Benedikt Nordhoff. Lock-sensitive interference analysis for java: Combining program dependence graphs with dynamic pushdown networks. In *Proc. 1st International Workshop on Interference and Dependence*, January 2013.
14. Christian Hammer. *Information Flow Control for Java.* PhD thesis, Universität Karlsruhe (TH), 2009.
15. Christian Hammer. Experiences with PDG-based IFC. In F. Massacci, D. Wallach, and N. Zannone, editors, *Proc. ESSoS'10*, volume 5965 of *LNCS*, pages 44–60. Springer-Verlag, February 2010.
16. Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6), December 2009.
17. Ben Hardekopf and Calvin Lin. Semi-sparse flow sensitive pointer analysis. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 226–238, New York, NY, USA, 2009. ACM.
18. S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proc. POPL '88*, pages 146–157, New York, NY, USA, 1988. ACM.
19. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.
20. M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *Proc. Formal Verification of Object-Oriented Systems*, 2011.
21. Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *Proc. 19th CSFW*, page 3. IEEE, 2006.
22. Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL '06*, pages 79–90. ACM, 2006.
23. Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proc. ESEC/FSE-11*, pages 178–187, New York, NY, USA, 2003. ACM.
24. Jens Krinke. Program slicing. In *Handbook of Software Engineering and Knowledge Engineering*, volume 3: Recent Advances. World Scientific Publishing, 2005.
25. Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A framework for the cryptographic verification of java-like programs. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th.* IEEE Computer Society, June 2012.
26. Lin Li and Clark Verbrugge. A practical MHP information analysis for concurrent Java programs. In *Proc. LCPC'04*, volume 3602 of *LNCS*, pages 194–208. Springer, 2004.
27. Andreas Lochbihler. Java and the Java memory model – a unified, machine-checked formalisation. In Helmut Seidl, editor, *Proc. ESOP '12*, volume 7211 of *LNCS*, pages 497–517, March 2012.
28. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.
29. Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *CSF*, pages 218–232, 2011.
30. Heiko Mantel and Henning Sudbrock. Flexible scheduler-independent security. In *Proc. ESORICS*, volume 6345 of *LNCS*, pages 116–133, 2010.
31. Heiko Mantel and Henning Sudbrock. Types vs. pdgs in information flow analysis. In *LOPSTR*, pages 106–121, 2012.
32. Heiko Mantel, Henning Sudbrock, and Tina Kraußer. Combining different proof techniques for verifying information flow security. In *Proc. LOPSTR*, volume 4407 of *LNCS*, pages 94–110, 2006.
33. Mangala Gowri Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to Java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, 2006.
34. Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proc. ESEC/FSE-7*, volume 1687 of *LNCS*, pages 338–354, London, UK, 1999.
35. Tri Minh Ngo, Mariëlle Stoelinga, and Marieke Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. In *ESSoS*, pages 107–122, 2013.

36. Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Trans. Program. Lang. Syst.*, 29(5):27, 2007.
37. Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proc. FSE '94*, pages 11–20, New York, NY, USA, 1994. ACM.
38. Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report 777, Computer Sciences Department, University of Wisconsin-Madison, 1988.
39. A. W. Roscoe, Jim Woodcock, and L. Wulf. Non-interference through determinism. In *ESORICS*, volume 875 of *LNCS*, pages 33–53, 1994.
40. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
41. Andrei Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proc. 5th International Andrei Ershov Memorial Conference*, volume 2890 of *LNCS*, Akademgorodok, Novosibirsk, Russia, July 2003.
42. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. CSFW '00*, page 200, Washington, DC, USA, 2000. IEEE Computer Society.
43. Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.
44. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. POPL '98*, pages 355–364. ACM, January 1998.
45. Gregor Snelting. Combining slicing and constraint solving for validation of measurement software. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 332–348, London, UK, 1996. Springer-Verlag.
46. Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.
47. Tachio Terauchi. A type system for observational determinism. In *CSF*, pages 287–300, 2008.
48. Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999.
49. Daniel Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.
50. Daniel Wasserrab. Information flow noninterference via slicing. *Archive of Formal Proofs*, 2010, 2010.
51. Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-based noninterference and its modular proof. In *Proc. PLAS '09*. ACM, June 2009.
52. Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *Proc. ISSTA*, pages 185–195. ACM, 2007.
53. Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. CSFW*, pages 29–43. IEEE, 2003.

Appendix A: Proof Sketch for Theorem 1 and 2

In the following, we describe the central steps in the soundness proof. All details can be found in [8].

Theorem 1. A program is low-security observational deterministic if

1. no low-observable operation o is potentially influenced by an operation reading high input,
2. no low-observable operation o is potentially influenced by a data conflict, and
3. there is no order conflict between any two low-observable operations.

Proof. Let two low-equivalent inputs be given. We have to demonstrate that, under conditions 1. – 3., all possible traces resulting from these inputs are low equivalent. The proof proceeds in a sequence of steps.

1. Definition. For a trace T and operation o, the *trace slice* $S(o,T)$ consists of all operations and dependences in T which form a path from *start* to o (see Figure 4). $S(o,T)$ is thus similar to a dynamic backward slice for o. Similarly the *data slice* $D(o,T)$ is the dynamic backward slice which considers only dynamic data dependencies, but not control dependencies. Trace and data slices are cycle free. Note that every operation in $S(o,T)$ has exactly one predecessor on which it is control-dependent, the *start* operation being the only exception. Note also that $S(o,T)$ can be soundly approximated by a static slice on $stmt(o)$, the source code statement containing o.

2. Lemma. Let q and r be two different operations of the same thread, and let T and U be two traces which both execute q and r. Further, let T execute q before r. Then U also executes q before r. This is a consequence of the fact that any dynamic branching point b imposes a total execution order on all operations $\in DCD(b)$, because according to 1., all operations have at most one control predecessor.

3. Lemma. Let q, r be operations which cannot happen in parallel, and let T and U be traces which both execute q and r. Further, let T execute q before r. Then U also executes q before r. Indeed if q, r are in the same thread, this is just the last lemma. Otherwise, MHP guarantees q executes before r's thread is forked, or r executes after q's thread has joined. Hence U executes q before r.

4. Lemma. Let $(\overline{m}, o, m)$ be a configuration in trace T. $\overline{T_o} = \overline{m}|_{use(o)}$ denotes the part of memory $\overline{m}$ that contains the variables used by o, and $T_o = m|_{def(o)}$ denotes the part of memory m that contains the variables defined by o. Now let T and U be two traces with low-equivalent inputs. Let o be an operation. If $D(o,T) = D(o,U)$ and no operation in these data slices reads high input, then $\overline{T_o} = \overline{U_o}$ and $T_o = U_o$. This lemma is proved by induction on the structure of $D(o,T)$ (remember $D(o,T)$ is acyclic).

5. Corollary. Let T, U be two traces with low-equivalent inputs. Let o be an operation. If $S(o,T) = S(o,U)$ and no operation in these trace-slices reads high values, then $\overline{T_o} = \overline{U_o}$ and $T_o = U_o$. That is, the low memory parts in both traces are identical for low-equivalent inputs, if all operations do not depend on high values.

6. Lemma. Let T and U be two finite traces of p with low-equivalent inputs. T and U are low-equivalent if for every low-observable operation o, $S(o,T) = S(o,U)$ holds and no operation in the trace-slices depends on high values, and T and U execute the same low-observable operations in the same relative order. This lemma, which seems quite natural, gives us an instrument for finite traces to prove the low-equivalence of traces resulting from low-equivalent input, which is necesasary for theorem 1. The infinite cases are treated in the next two lemmata.

7. Lemma. Let T and U be two infinite traces of p with low-equivalent inputs such that $obs_{low}(T)$ is of equal length or longer than $obs_{low}(U)$ (switch the names if necessary). T and U are low-equivalent if

- they execute the shared low-observable operations in the same relative order,
- for every low-observable operation $o \in U$ $S(o,T) = S(o,U)$ holds and no operation in the trace-slices reads high input
- and for every low-observable operation $o \in T$ and $o \notin U$ U infinitely delays an operation $b \in DCD(o)$.

8. Lemma. Let T and U be two traces of p with low-equivalent inputs, such that T is finite and U is infinite. T and U are low-equivalent if

- $obs_{low}(T)$ is of equal length or longer than $obs_{low}(U)$,
- T and U execute the shared low-observable operations in the same relative order,
- for every low-observable operation $o \in U$ $S(o,T) = S(o,U)$ holds and no operation in the trace-slices reads high input
- and for every low-observable operation $o \in T$ and $o \notin U$ U infinitely delays an operation $b \in DCD(o)$.

9. Corollary. Traces T, U are low-equivalent if one of the last three lemmata can be applied. What remains to be shown is that the preconditions of the lemmata are a consequence of the conditions 1. – 3. in theorem 1.

10. Lemma. If operation o is not potentially influenced by a data conflict, then $S(o,T) = S(o,U)$ holds for all traces T and U which execute o. Note that only at this point, data or order conflicts are exploited. This lemma needs an induction over the length of T. The base case is trivial, because both T, U consist only of the *start* operation, and trivially $S(start,T) = S(start,U)$. For the induction step, let q be the next operation in T. If $o \notin DFS(q)$, then $q \notin S(o,T)$, and the induction step trivially holds. Otherwise, one can show that every dynamic data or control dependence $r \overset{v}{\dashrightarrow} q$ and $r \overset{dcd}{\dashrightarrow} q$ in $S(q,T)$ is also in $S(q,U)$. Furthermore, q does not depend on additional operations in U. Thus q has the same incoming dependences in T and U. By induction hypothesis, $S(r,T) = S(r,U)$ for every r on which q is dependent in T and U. Hence $S(q,T) = S(r,T)$.

11. Lemma (see section 4.3, lemma 1). Let o be an operation that is not potentially influenced by a data conflict or an operation reading high input. Let T be a trace and Θ be the set of possible traces whose inputs are low-equivalent to the one of T. If $o \in T$, then every $U \in \Theta$ either executes o or infinitely delays an operation in $DCD(o)$.

12. Lemma. Let T and U be two traces with low-equivalent inputs. If there are no order conflicts between any two low-observable operations, then all low-observable operations executed by both T and U are executed in the same relative order.

13. Theorem 1 holds. Lemma 12 guarantees that T and U execute the shared low-observable operations in the same relative order. Lemma 11 can be applied to all low-observable operations o executed by both T and U, hence $S(o,T) = S(o,U)$. Since the potential influence of a low-observable operation o does not contain operations reading high input, this also holds for the operations in $S(o,T)$ and $S(o,U)$. To prove low-equivalence of T and U, we apply one of the lemmata 6,7, or 8, depending whether T resp. U are finite or infinite.

Remember that the three conditions for theorem 1 can naturally be checked using PDGs and slicing. This fact justifies our definition of low-equivalent traces, and our PDG-based approach.

Theorem 2. If a program is LSOD according to definition 6, it is probabilistically noninterferent.

Proof. We write Θ_i for the set of possible traces for input i; for a trace $r \in \Theta_i$, $P_i(r)$ is its execution probability. Now let two low-equivalent inputs t, u be given. Let $\Theta = \Theta_t \cup \Theta_u$. Let $T \in \Theta$. Let $\mathfrak{T} = \{r \in \Theta_t \mid r \sim_{low} T\}$, $\mathfrak{U} = \{r \in \Theta_u \mid r \sim_{low} T\}$. We have to show that $\sum_{r \in \mathfrak{T}} P_t(r) = \sum_{r \in \mathfrak{U}} P_u(r)$ if LSOD holds.

Due to LSOD, all traces in Θ and thus in $\mathfrak{T} \cup \mathfrak{U}$ are low-equivalent, and $\forall r \in \Theta : T \sim_{low} r$. Therefore, under LSOD, $\mathfrak{T} = \Theta_t$, because the condition $T \sim_{low} r$ in the definition of $\mathfrak{T}$ always holds. That is, $\mathfrak{T}$ contains *all* possible traces for inputs t, u. Therefore $\sum_{r \in \mathfrak{T}} P_t(r) = 1$. Similarly, $\sum_{r \in \mathfrak{U}} P_u(r) = 1$, and the required equality holds. QED.