**Karlsruhe Reports in Informatics 2012, 6**
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

# Probabilistic Noninterference Based on Program Dependence Graphs

Dennis Giffhorn
Gregor Snelting

Institute for Program Structures and Data Organization

2012

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

# Probabilistic Noninterference Based on Program Dependence Graphs*

Dennis Giffhorn, Gregor Snelting
Lehrstuhl Programmierparadigmen, Karlsruher Institut für Technologie

April 23, 2012

## Abstract

We present a new algorithm for checking probabilistic noninterference in concurrent programs. The algorithm uses the *Low-Security Observational Determinism* criterion. It is based on program dependence graphs for concurrent programs, and is thus very precise: it is flow-sensitive, context-sensitive, object-sensitive, and optionally time-sensitive. The algorithm has been implemented for full Java bytecode with unlimited threads, and avoids restrictions or soundness leaks of previous approaches. A soundness proof is provided. Precision and scalability have been experimentally validated.

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

## 1 Introduction

Information flow control (IFC) discovers software security leaks by analysing the software source or machine code. IFC for concurrent or 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 program in the middle 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 typical probabilistic leak.

IFC aims at discovering all such security leaks. Most IFC analyses check some form of noninterference [28], and to this aim classifies program variables, input and output as high (secret) or low (public). *Probabilistic Noninterference* (PN) [32, 30, 29, 31, 19] is the established security criterion for concurrent programs. It is difficult to guarantee PN, as an IFC must in principle check all possible interleavings and their impact on execution probabilities. This is why some PN analyses put severe restrictions on program or scheduler behaviour.

One specific form of PN however is scheduler independent: *Low-Security Observational Determinism* (LSOD) demands that for a program which runs on two low-equivalent inputs, all possible traces are low-equivalent [21, 27, 40, 14]. Low-equivalent input streams coincide on low input values, and low-equivalent traces coincide on operations on low variables resp. low memory cells. Earlier research [40, 14] has shown that the LSOD criterion guarantees PN, and that LSOD can be implemented as a program analysis. The following criterion is sufficient to guarantee LSOD [40]: 1. program parts contributing to low-observable behaviour are conflict-free, that is, the

[*] This article is based on parts of the PhD thesis of the first author [6], with additional contributions by the second author. The research was partially supported by Deutsche Forschungsgemeinschaft (DFG grant Sn11/9-2) and in the scope of the priority program "Reliably Secure Software Systems" (DFG grant Sn11/12-1).

```
void main():
  x = inputPIN();
  if (x < 1234)
     print(0);
  y = x;
  print(y);
```

```
void thread_1():
  x = input();
  print(x);

void thread_2():
  y = inputPIN();
  x = y;
```

```
void thread_1():
  x = 0;
  print(x);

void thread_2():
  y = inputPIN();
  while (y != 0)
     y--;
  x = 1;
  print(2);
```

Figure 1: Examples for information leaks in sequential programs (left), for a possibilistic channel (mid) and for probabilistic channels (right).

program is low-observable deterministic; 2. implicit or explicit flows do not leak high data to low-observable behaviour. Several attempts to devise analysis algorithms for LSOD however turned out to be unsound, unprecise, or very restrictive. Hence LSOD never gained popularity, and there are no implementations for realistic languages.

It is the main contribution of this paper to demonstrate that the LSOD criterion can be checked easily and naturally using *Program Dependence Graphs* (PDGs) and slicing algorithms for concurrent programs. PDGs have already been developed as an IFC analysis tool for full sequential Java [11, 10, 33, 38], and demonstrated high precision and scaleability. In the current article, we show how to use PDGs for a precise LSOD checker. Our LSOD checker uses a new definition of low-equivalent traces, which – in case of infinite (i.e. 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. The algorithm has been implemented for full Java including an arbitrary number of threads. It also exploits advances in the may-happen-in-parallel (MHP) analysis of concurrent programs, which allow even time-sensitive and lock-sensitive MHP and IFC. We present details of the algorithm, a soundness proof, and empirical data about precision and scalability.

# 2 Dependence Graphs and Noninterference

## 2.1 PDG Basics

Program dependence graphs are a standard tool to model information flow through a program. Program statements or expressions are the graph nodes. There are two kinds of edges: data dependences and control dependences. A data dependence edge $x \rightarrow y$ means that statement $x$ assigns a variable which is used in statement $y$, without being reassigned underway. A control dependence edge $x \rightarrow y$ 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). Control and data dependences are transitive. For a dependency $x \rightarrow y$, $x$ is called the source and $y$ the sink of the dependency.

In a PDG $G = (N, \rightarrow)$, a path $x \rightarrow^* y$ means that information can flow from $x$ to $y$; if there is no path, it is guaranteed that there is no information flow [12, 26, 24, 38]. Thus PDGs are *correct*. Exploiting this fundamental property, all statements possibly influencing $y$ (the so-called *backward slice*) are easily computed as $BS(y) = \{x \mid x \rightarrow^* y\}$. $y$ is called the *slicing criterion* of the backward slice. Similarly, the forward slice is $FS(x) = \{y \mid x \rightarrow^* y\}$. The Slicing Theorem [26] shows that for any initial state on which the program terminates, the program and its slice compute the same sequence of values for each element of the slice.

As an example, consider the small program and its de-

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

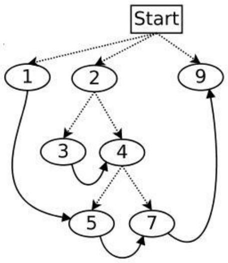

Figure 2: A small program and its dependence graph

pendence graph in figure 2. Control dependences are shown as dotted edges, data dependences are shown as solid edges; transitive dependences are not explicitly part of the PDG. 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.

PDGs and backward slices for realistic languages with procedures, complex control flow, and data structures are much more complex than the basic concept sketched above. Our interprocedural analysis is based on the context-sensitive Horwitz-Reps-Binkley (HRB) algorithm [25, 13], which uses so-called summary edges to model flow through procedures. Full sequential Java requires even more complex algorithms, which have been described in [8, 11]. Thus in the general case, computation of $BS(x)$ involves much more than just backward paths in the PDG.

The power of PDGs stems from the fact that they are flow-sensitive, context-sensitive, and object-sensitive: the order of statements does matter and is taken into account, as is the actual calling context for procedures, and the actual reference object for method calls. Thus the PDG resp. backward slice never indicates influences which are in fact impossible due to the given statement execution order of the program; only so-called "realizable" (that is, dynamically possible) paths are considered. But this precision is not for free: PDG construction can have complexity $O(|N|^3)$. In practice, PDG use is limited to programs of about 100kLOC [2].

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

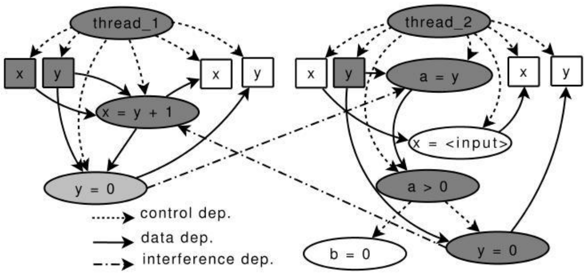

Figure 3: Interference dependences between two threads.

## 2.2 Noninterference and PDGs

As explained, a missing path from $a$ to $b$ in a PDG – or more precisely $a \notin BS(b)$ – 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 channels such as timing leaks. It is therefore not surprising that noninterference is related to PDGs.

Indeed we have published 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 [38, 36].

## 2.3 PDGs and Slicing for Multi-Threaded Programs

For multi-threaded programs operating on shared memory, PDGs are extended by so-called *interference dependencies*[1] which connect an assignment to a variable in one thread with a read of the same variable in another thread [15]. Figure 3 shows a small example with two interference edges.

The simplest slicer for concurrent programs is the iterated two-phase slicer (I2P) [22, 9]. I2P uses the context-

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

sensitive HRB algorithm[2], 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. Due to locks or the fork/join structure, not all statements can be executed in parallel. In fact, often a program analysis can prove that certain statements can *not* 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. [23, 16, 6].

Unfortunately, I2P is – even with MHP – not very precise, 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 3 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 gray node `y = 0;` in `thread_1` cannot influence the node `x = y+1;`. The example also demonstrates that interference dependences are – in contrast to data and control dependencies – 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, as summary edges are transitive. This explains why I2P needs a worklist for interference edges.

A *time-sensitive analysis* discards impossible, "time-travelling" flows.[3] Time-sensitivity is based on the fork/join structure of threads, and is even more complex for interprocedural analysis. I2P is context-sensitive inside threads, but not time-sensitive; benchmarks have shown that its precision (i.e. slice size) is ca. 25% worse than the best known time-sensitive slicer [6].

Figure 4 shows an example for time-sensitive slicing and MHP. MHP prunes the interference dependence 20 → 13, as node 20 is always executed after node 13. Furthermore, the time-sensitive backward slice for node 14 does not include nodes 16 - 20: in order to influence node 14 via node 9 and 13, node 20 has to be executed before nodes 9 and 13. But thread 2 is started only after node 13, hence the dependence chain 20 → 9 → 13 → 14 is time-sensitively impossible. The example also shows that time-sensitivity is improved by MHP: without MHP, edge 13 → 20 would exist and force nodes 16 – 20 into the time-sensitive slice.

Time-sensitive slicing algorithms are very complex, and cannot be described in detail here. Indeed, for many years, no scalable algorithm existed. Today, our new algorithms, which are based on earlier work by Krinke and Nanda [15, 22] allow to analyse at least a few kLOC of full Java. The algorithms can handle full Java with an arbitrary number of threads, and are described in detail in [5, 7, 6].

# 3 Formalising LSOD

## 3.1 Low-equivalent traces

Before presenting a formal definition of our LSOD criterion, let us begin with an informal description. We assume that program data, input values, output values, and statements are classified either low (public) or high (secret). Note that in our flow-sensitive approach, classification of values and variables happens *per statement* – there is no global classification of variables or memory cells. Depending on the context, a variable may at one program point contain a low value, and at another point a high value. Still, soundness is guaranteed, while flow-sensitivity (which is naturally provided by PDGs) offers precision gains and less restrictions on programs (see also section 6). Note that most of the statement-level classifications can be derived automatically from classification of just program inputs and outputs by a fixpoint iteration on the PDG [11].

We assume that a potential attacker knows the source code and its PDG, and can observe execution of all op-

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

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

```
int x, y;

void main():
  x = 0;
  y = 1;
  fork thread_1();
  int p = x - 2;
  int q = p + 1;
  y = q * y;
```

```
void thread_1():
  int a = y + 1;
  fork thread_2();
  int b = a * 4;
  x = b / 2;

void thread_2():
  y = 0;
```

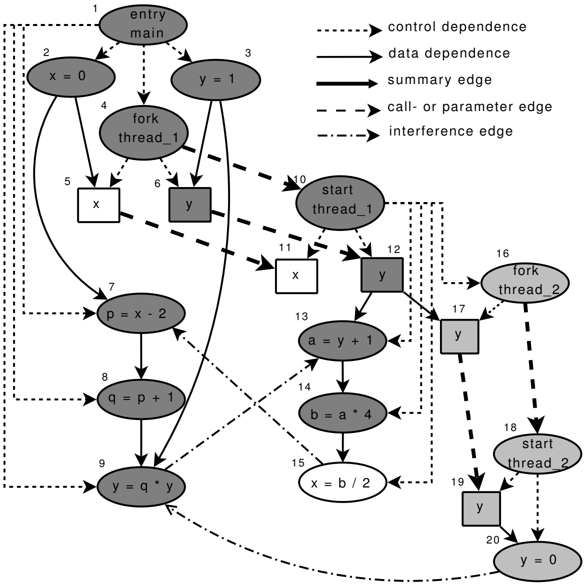

Figure 4: More precise MHP information results in more precise slices: The gray nodes denote the time-sensitive slice for node 14 in case all threads are assumed to happen in parallel. The dark gray nodes denote the slice after MHP analysis, which prunes edge 20 → 13, which removes nodes 16-20 from the slice.

erations (i.e. dynamically executed statements) and their operands which read or write a low value. The attacker can however not observe high values, or high operations. For example, if the source statement `print(x)` is classified low, then its dynamic execution and printed value can be observed; if `read(x)` is classified high, the dynamically read value of `x` cannot be observed. Remember that variable `x` does not posses a global classification; correspondingly, the attacker can *not* see all low values at any time (see also discussion in section 6).

Of course, the attacker can also see low input; we assume that program input is a list of (perhaps non-primitive) values, where each value is classified high or low. Inputs are low-equivalent if the sublists of low input values are equal. The attacker is also aware of the probability with which an input causes a certain low-observable behavior. Under these assumptions, our LSOD criterion guarantees to discover all explicit, implicit, possibilistic, or probabilistic leaks. Note that our LSOD does not check termination leaks, for reasons to be discussed later.

Technically, 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 of the operation, and the memory after executing it. Note that a trace is valid only with respect to a specific interleaving or scheduling of program threads. Low-observable events are configurations from a trace which read or write low memory cells; only low memory cells are part of low-observable events. The *low-observable behaviour* of an execution trace is the subtrace which contains only low-observable events. *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 in a way similar to classical (sequential) noninterference; using program traces instead of program states. This also explains why LSOD is scheduler independent.

The natural definition of low-equivalent traces becomes however problematic in case of nontermination. The classical definition, as formalized in the literature, allows one trace to terminate and the other to not terminate. Hence the problem of *infinite traces* and *termination channels* arises, which needs to be discussed before we formally present our new definition of low-equivalent traces.

Consider the program in the middle of figure 5, whose

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

Figure 5: Three tough nuts for termination-insensitive definitions of low-equivalent traces. The program on the left must be rejected because it gradually leaks the PIN, the one in the mid could be accepted because its leak is a termination channel. The program on the right exploits termination channels to leak the input PIN.

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 5 (right) exploits a termination channel which leaks the PIN completely. It is known that in case of interactive programs, termination channels can leak arbitrary amounts of information [1]. To prevent this, several variants of LSOD resp. PN forbid low-observable events behind loops guarded by high data.

However, this is a severe, if not unacceptable restriction in practice. Sometimes, program analysis can deduce that a loop will terminate, and in such cases 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 discussion in section 6).

We thus aim at a sound formal definition of LSOD which however may allow termination leaks. This approach has already been tried in earlier research. Published LSOD definitions that aim to permit termination channels declare traces to be low-equivalent if their low-observable behavior is equal up to the length of the shorter sequence of low-observable events [34, 40]. But as pointed out by Huisman et al. [14], this may lead to unintended information leaks. Consider the program on the left side of figure 5, whose traces always diverge, and assume that the input PIN is high data and that the executions of the print statement comprise its low-observable behavior. 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 behavior is perfectly legal, because all traces with low-equivalent inputs are equal up to the length of the shorter sequence (see more detailed discussion in section 6).

In order to solve this problem, we suggest the following new approach: For finite traces, we stick to the common definition that they are low-equivalent if their low-observable behaviors are equal. If both traces are infinite, then the low-observable behaviors must be equal up to the length of the shorter sequence, **and** the low-observable events missing in the other trace must be missing due to infinite delay. The latter additional constraint is new and makes sure that the missing events leak information only via termination channels. Similarly, if one of both traces is finite and the other is infinite, then the finite trace must have *at least as much* low-observable events as the infinite one, the low-observable behaviors 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 infinite delay.

Fortunately, our new constraint can be statically approximated through PDGs and slicing. Without a PDG approach, our new and – as we will argue later – more powerful definition of LSOD would not be useable.

```
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);
```

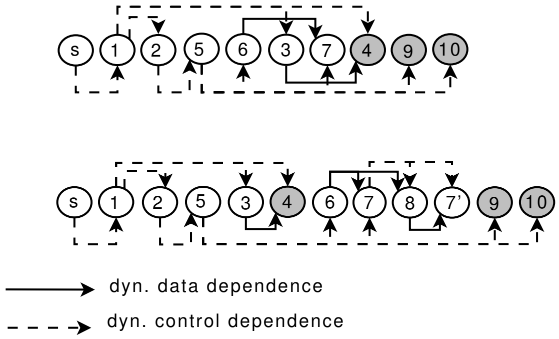

Figure 6: A program and two possible traces. The first trace results from input (inputPIN() = 0, input() = 0), the second from (inputPIN() = 1, input() = 0). The shaded nodes represent the low-observable behavior.

## 3.2 Formalizing low-equivalent traces and LSOD

In the following, we will formally develop our definition, prove that it guarantees LSOD, and use it as basis for an algorithm. Full details can be found in [6].

**Definition 1 (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*

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

*is a list of events*

$$obs_{low}(T) = event_{low}(\overline{m}_1, o1, m_1), \dots, event_{low}(\overline{m}_k, o_k, m_k)$$

*where*

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

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

We write $o \in T$ if $\exists \overline{m}, m : (\overline{m}, o, m) \in T$. Operations in (different) traces can be uniquely identified by their calling-context and control-flow history (see below). Operations which perform a low read or low write are low-observable, together with the corresponding parts of the memory. Note that $\overline{m}\mid_{use(o)}$ resp. $m\mid_{def(o)}$ do not contain high memory cells.

For later use within our LSOD criterion, traces must be enriched with dynamic control and data dependencies as in figure 6: 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 (similar to dynamic slicing [39]). This is formalized in

**Definition 2 (Dynamic dependencies)** *Let $T$ be a trace of a program $p$.*

1. *An 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 $o$ [39].*
2. *An operation $o$ is dynamically data dependent on operation $a$ in $T$, written $a \overset{v}{\dashrightarrow} o$, iff there exists a variable $v \in use(o) \cap def(a)$, $o$ executes after $a$ in $T$ and*

*there is no operation $o'$ with $v \in def(o')$ executing between $a$ and $o$ in $T$.*

3. *$Pot(o) = \{q \in T \mid o\ (\overset{v}{\dashrightarrow} \cup \overset{dcd}{\dashrightarrow})^*\ q\}$ denotes the set of all operations which are (transitively) dynamically control or data dependent on $o$.*

   *$DCD(o) = \langle q_1 \dots q_n \mid q_i \in T, q_1 = start, q_n = o, q_i \overset{dcd}{\dashrightarrow}^* o\rangle$ is the list of operations on which $o$ is (transitively) dynamically control (but not data) dependent; topologically sorted by dynamic dependency.*

Note that dynamic dependencies are cycle-free. $Pot(o)$, the operations potentially influenced by $o$, can be seen as a dynamic forward slice; $DCD$ can be seen as a dynamic backward control slice. In figure 6 (lower part), $Pot(5) = \{6,7,8,7',9,10\}$, and $DCD(4) = \langle start, 1, 4\rangle$.

It is important to note that every operation 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 $b = dcp(o)$.

Note also that dynamic dependencies and thus *Pot* and *DCD* can be soundly approximated by static slices. In particular if $Pot(o) = \{p_1, \dots, p_k, \dots\}$ then $\{stmt(p_1), \dots, stmt(p_k), \dots\} \subseteq FS(stmt(o))$. If $p \notin Pot(o)$, it is guaranteed that $o$ cannot influence $p$ through explicit or implicit flow.

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 formalization of the notion "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 explicitely 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.[4] 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 just means 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 3 (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 $U$ executes $b$ and chooses the same branch as in $T$, then either $U$ executes $o$, or does not execute $o$ due to e.g. an infinite loop between $b$ and $o$. This definition is used for the formalization of low-equivalent traces:

**Definition 4 (Low-equivalence of traces, $\sim_{low}$)** *Let $p$ be a program and let $T$ and $U$ be two traces of $p$. 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 behaviors. 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 $\forall 0 \leq i \leq k_T$ :*

$$\overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$$

[4] Note that exceptions and handlers generate additional control dependencies in PDGs and traces [11]. Thus if `o1` may throw an exception, the dependency situation is more complex than in a "regular" `if(b){o1;o2}`. Still, the following argument for traces holds.

2. *$T$ is finite and $U$ is infinite, and*
   - $k_T \geq k_U$,
   - $\forall 0 \leq i \leq k_U : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, *and*
   - $\forall k_U < j \leq k_T$, *$U$ infinitely delays an operation $b \in DCD(o_j)$.*

3. *$T$ is infinite and $U$ is finite, and*
   - $k_U \geq k_T$,
   - $\forall 0 \leq i \leq k_T : \overline{m}_i = \overline{n}_i \wedge o_i = q_i \wedge m_i = n_i$, *and*
   - $\forall k_T < j \leq k_U$, *$T$ infinitely delays an operation $b \in DCD(q_j)$.*

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

In this definition, "$U$ infinitely delays $b \in DCD(o_j)$" explicitely expresses that the delayed operation must not necessarily be the low-observable $o_j$, but can be a dynamic control predecessor of $o_j$. In any case, $b$ is on the dynamic control path between $o_{k_U}$ and $o_j$.

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

**Definition 5 (Low-security observational determinism)** *Program $p$ 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.*

Zdancewic and Myers [40] 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 starting point for our own work, as we realized that not only explicit and implicit flow can naturally be checked using PDGs [11], but also conflicts and their impact are naturally modeled in PDGs enriched with conflict edges. We thus provide the following definition:

**Definition 6 (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.*
- *An operation $o$ is* potentially influenced by a data conflict *if there exists operations $a, b$ such that $o \in Pot(b)$ and $a \overset{dconf}{\rightsquigarrow} b$.*

The following lemma states that an operation in a trace which is not influenced by a data conflict or by high data, is either executed in all traces with low-equivalent input, or is delayed infinitely due to some nontermination in other operations on which it is control dependent.

**Lemma 1** *Let $p$ be a program. Let $T$ be a trace of $p$ and $\Theta$ 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 an operation $q$ reading high input: $o \notin (Pot(a) \cup Pot(b) \cup Pot(q))$. If $o \in T$, then every $U \in \Theta$ either executes $o$ or infinitely delays an operation in $DCD(o)$.*

Only this lemma justifies our definition of low-equivalent traces. Appendix A sketches the most important proof steps; the full proof can be found in [6]. Note again that dynamic dependencies can be soundly approximated using PDGs and slices; hence it can be statically checked whether an operation is missing due to infinite delay. This remarkable fact, which also guarantees high precision, characterizes the core difference between our definition and earlier approaches.

The fundamental soundness theorem for our LSOD criterion can now be stated:

**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.*

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.

The full proof can be found in [6]; in appendix A we explain the most important proof steps in a semiformal way. Note that the theorem is only valid if *sequential consistency* can be assumed. The Java memory model was designed to guarantee sequential consistency for race-free programs, and today formal definitions and machine-checked guarantees of the JMM are available [18, 17].

A detailed discussion and comparison of our LSOD variant with other LSOD variations from the literature is presented in section 6.

## 4 A Slicing-based LSOD Check

We will now show how to implement a flow-sensitive, context-sensitive, and optionally time-sensitive IFC which is based on theorem 1. Indeed, the three conditions of theorem 1 can naturally be checked using slicing for concurrent programs, as slicing provides sound and precise approximations of dynamic dependencies.

First, we assume that all PDG nodes are annotated (classified) with a security level. In practice it is enough to annotate program inputs and outputs, as the security level for intermediate nodes can be determined by a fixpoint iteration similar to data flow analysis on the PDG [11, 10]. Note also that the analysis can handle arbitrary lattices of security levels, not just the two-element lattice with levels “high” and “low”. The implementation offers even a lattice editor [9].

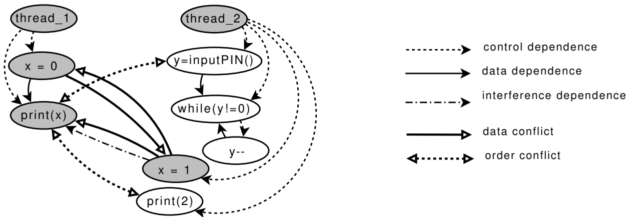

Figure 7: PDG of the program on the right side of Fig. 1, enriched with data and order conflict edges. The gray nodes denote the slice for node `print(x)`. Note that the slice ignores conflict edges.

For the implementation of the LSOD criterion using PDGs, these are enriched with data and order conflict edges. The resulting structure is called a CPDG (conflict-enriched program dependency graph).

**Definition 7 (Data and order conflict edges)** *Let m and n be two nodes in a PDG G that may happen in parallel. There is a data conflict edge* $m \to_{dconf} n$ *to G if m defines a variable v that is used or defined by n. There is an order conflict edge* $m \leftrightarrow_{oconf} n$ *to G if both nodes are classified as sources or sinks.*

Figure 7 shows the CPDG of the program on the right side of Fig. 1, enriched with conflict edges. 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)`.

The algorithm also needs

**Definition 8 (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 [6]. Once CPDG and TCFG have been constructed, the IFC checker proceeds as follows:

1. Compute a backward slice for every sink (program output) *s*. Let *l* be the security level of *s*.

2. If the backward traversal encounters a source of level $h \not\sqsubseteq l$, then the program may leak data of level $h$ via explicit or implicit flow and is rejected. Note that this criterion is also used in our sequential IFC [11].

3. If the traversal encounters an incoming data conflict edge, the program may contain a probabilistic data channel and is rejected.

4. If the traversal encounters an order conflict edge, check if the order conflict is low-observable (i.e. both conflicting nodes are classified *low*). If so, the program may contain a probabilistic order channel and is rejected.

As an example, consider Fig. 7. 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 gray in figure 7, 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 avoid false alarms in programs which do not use any high data, we optimize item 4 in the above algorithm as follows:

4. If the traversal encounters an order conflict edge, check if the order conflict is low-observable. If so, check in the TCFG if any node that can be executed before both conflicting nodes is a high source. If so, the program may contain a probabilistic order channel and is rejected.

In the implementation, rules 2. - 4. are integrated into a backward slicer for multi-threaded Java. Note that we use an I2P slicer: a time-sensitive slicer would be more precise, but would make the algorithm much more complex and expensive. Integration of a time-sensitive slicer is left for future work. 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 too many false alarms.

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 [11] is used). If the program passes that check, it is scanned for probabilistic channels by checking rules 3 and 4. This is done by Alg. 2.

**Algorithm 1** Information flow control for concurrent programs.

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

  /* Check implicit and explicit flow: */
  Let $\texttt{flow}(G, C, \mathfrak{L})$ be a function that returns false if $G$ contains illicit implicit or explicit flow.
  if $\texttt{flow}(G, C, \mathfrak{L}) == \texttt{false}$ then
    return false

  /* Scan the program for probabilistic channels. */
  /* Check sources: */
  for all $n \in N : \texttt{src}(n) \neq \bot$ do
    if $\texttt{prob}(G, C, n, \texttt{src}(n), \mathfrak{L}) == \texttt{false}$ then
      return false
  /* Check sinks: */
  for all $n \in N : \texttt{sink}(n) \neq \top$ do
    if $\texttt{prob}(G, C, n, \texttt{sink}(n), \mathfrak{L}) == \texttt{false}$ then
      return false

  return true
```

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.

**Algorithm 2** Procedure `prob` detects probabilistic channels.

```
Require: An CPDG $G = (V,E)$, its TCFG $C$, a node $s$,
  its security level $l$, the security lattice $\mathfrak{L}$.
Ensure: '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 $\texttt{benign}(C,m,n,oconf,\mathfrak{L},x) == \texttt{false}$ 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
    $W = W \setminus \{n\}$ {remove next node n from W}
    /* look for data conflicts */
    for all $m \rightarrow_{dconf} n$ do
      if $\texttt{benign}(C,m,n,dconf,\mathfrak{L},l) == \texttt{false}$ then
        /* conflict is harmful */
        return false
    /* proceed with standard I2P slicing */
    /* handle incoming edges, exclude conflict edges */
    for all $m \rightarrow_e n, e \notin \{oconf, dconf\}$ do
      /* if m hasn't been visited yet or we are in phase 1
      and m has been visited in phase 2 */
      if $m \notin \text{dom } M \vee (\neg M(m) \wedge (M(n) \vee e == conc))$
      then
        /* if we are in phase 1 or if e is not a call or
        param-in edge, add m to W */
        if $M(n) \vee e \notin \{pi, call\}$ then
          $W = W \cup \{m\}$
        /* determine how to mark m: */
        if $M(n) \wedge e == po$ then
          /* we are in phase 1 and e is a param-out
          edge: mark m with phase 2 */
          $M = M \cup \{m \mapsto false\}$
        else if $\neg M(n) \wedge e == conc$ 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 $W = \varnothing$
  return true /* no probabilistic channels */
```

**Algorithm 3** Procedure `benign` identifies benign conflicts.

```
Require: 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 $l \in \mathfrak{L}$.
Ensure: 'true' if the conflict is harmless, 'false' otherwise.
  Let $\texttt{reaches}(m,n,C)$ 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
    $x = (\texttt{src}(a) \neq \bot \wedge \texttt{src}(a) \sqsubseteq l) \vee (\texttt{sink}(a) \neq \top \wedge \texttt{sink}(a) \sqsubseteq l)$ {is 'a' visible?}
    $y = (\texttt{src}(b) \neq \bot \wedge \texttt{src}(b) \sqsubseteq l) \vee (\texttt{sink}(b) \neq \top \wedge \texttt{sink}(b) \sqsubseteq l)$ {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 $\texttt{src}(n) \not\sqsubseteq l$ then
      if $(\texttt{reaches}(n,a) \vee n \parallel a) \wedge (\texttt{reaches}(n,b) \vee n \parallel b)$
      then
        return false /* the conflict is harmful */

  return true /* the outcome of the conflict cannot be
  influenced by high data */
```

Procedure `benign` 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. Next, it checks for both order and data conflicts whether the involved nodes can be preceded by a source $n$ of high data. This is the case if $n$ reaches them on realizable paths in the TCFG or if it may happen in parallel to them.

**Example.** Let us apply the algorithm to figure 7. 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)`. Just for illustration, let us trace 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`. But for illustration, let us assume this order conflict would not exist, and 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)`, so algorithm 3 is called, which discovers that a source of high data, namely `y=inputPIN();` can reach `print(x);` hence the data conflict is harmful. The I2P slicer does not preceed any further, but immediately returns `false`.

The other two calls to algorithm 2 from algorithm 1 proceed similarly. The example shows that the LSOD check is terminated as soon as a leak is found; it can also be modified to return a list of *all* leaks (CPDG paths).

**Run-time analysis.** Let us now discuss the run-time of our algorithm. If we exclude the check of the implicit and explicit flow done by the external procedure `flow`, the runtime complexity is dominated by the reachability check in procedure `benign`. For each conflict edge, $\mathscr{O}(|N_{CPDG}|)$ complete traversals of the TCFG may be necessary, and the number of conflict edges in the CPDG is bound by $\mathscr{O}(|N_{CPDG}|^2)$. This means a worst case complexity of $\mathscr{O}(|E_{CPDG}| + |N_{CPDG}|^3 * |E_{TCFG}|)$ for one call of procedure `prob`, the first summand being the upper bound for the costs of the extended I2P slicer, the second summand, for procedure `benign`. Since `prob` can be called $\mathscr{O}(|N_{CPDG}|)$ times in the worst case, the complete IFC check has a worst case complexity of $\mathscr{O}(|N_{CPDG}| * |E_{CPDG}| + |N_{CPDG}|^4 * |E_{TCFG}|)$.

Several optimizations trading memory for speed are possible. The harmlessness of a conflict edge needs to be checked at most once for each security level in the lattice $\mathfrak{L}$ and can be cached and reused. It also can be precomputed which sources of information can reach which conflicting nodes, which can be efficiently done in $\mathscr{O}(|N_{CPDG}| * |E_{TCFG}|)$ by computing a context-sensitive forward slice of the TCFG for each source. Both optimizations together reduce the runtime complexity to $\mathscr{O}(|N_{CPDG}|)$ slices of the CPDG in `prob`, $\mathscr{O}(|\mathfrak{L}| * |N_{CPDG}|^2)$ calls of `benign` and $\mathscr{O}(|N_{CPDG}|)$ slices of the TCFG, summing up to a total complexity of $\mathscr{O}(|N_{CPDG}| * |E_{CPDG}| + |\mathfrak{L}| * |N_{CPDG}|^2 + |N_{CPDG}| * |E_{TCFG}|)$.

# 5 Evaluation

The above algorithms have been implemented for full Java, and integrated into the sequential IFC analysis as described in [11]. In the following, we apply the algorithm to two example programs, and we evaluate scalability through a benchmark. More case studies can be found in [6]. To our knowledge, no other evaluations of LSOD precision or scaleability have been published, hence we cannot compare our implementation to other algorithms.

## 5.1 Analysis of literature examples

Our first example program is from [32], and is given in figure 8. 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`, so an IFC analysis considering only these kinds of information flow classifies the program secure. But 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 actually, according to [32], if the input `PIN` is less twice the value of variable `mask`, then `PIN`'s value is eventually copied into `result` and printed to the screen (provided that scheduling is fair).

We classified statement `PIN = Integer.parseInt (args[0])` as a high source and `System.out.println (result)` as a low sink. No other classifications were necessary. Our algorithm detected a probabilistic data channel from the source to the sink, as required.

Our second example is from [20], and is given in figure 9. The program is probabilistic noninterferent, hence safe, which is however difficult to discover: only very precise IFC will avoid false alarms. The program manages a stock portfolio of Euro Stoxx 50 entries. It consists of four threads, coordinated by an additional `main` thread. The program first runs the `Portfolio` and `EuroStoxx50` threads concurrently, where `Portfolio` reads the user's stock portfolio from storage and `EuroStoxx50` retrieves the current stock rates. When these threads have finished, threads `Statistics` and `Output` are run concurrently, where `Statistics` calculates the current profits and `Output` incrementally prepares a statistics output. Af-

```
  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);
    }
  }
```

Figure 8: Example from Smith and Volpano [32]

ter these threads have finished, the statistics are displayed, together with a pay-per-click commercial. An ID of that commercial is sent back to the commercials provider to avoid receiving the same commercial twice. The portfolio data, `pfNames` and `pfNums`, is secret, hence the Euro Stoxx request by `EuroStoxx50` and the message sent to the commercials provider should not contain any information about the portfolio. As `Portfolio` and `EuroStoxx50` do not interfere, 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.

Our MHP analysis was able to detect that the joins of the threads are must-joins, which enabled our IFC algorithm to identify that there is no order conflict between the two flushes of `nwOutBuf`, therefore no probabilistic channel was reported. This example thus demonstrates the high precision of our algorithm.

## 5.2 Runtime Behavior

We investigated how well our implementation scales with increasing program sizes, lattice sizes and numbers of sources and sinks. We applied our algorithm to a benchmark of 8 small and medium-sized programs between 200 and 3000 LOC (taken from the Bandera benchmark and the JavaGrande benchmark). We used three different security lattices: Lattice A is a simple chain of three elements, *public* $\sqsubseteq$ *confidential* $\sqsubseteq$ *secret*. Lattice B consists of 22 elements, arranged in a lattice of height 9, resulting in many incomparable pairs of elements. Lattice C

```
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;
  }
}
```

```
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];
    }
  }
}
```

Figure 9: Example from Mantel et al [20], converted to Java. For brevity, some methods are not shown.

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

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

is a huge lattice of height 7 with 254 elements. For each program and lattice, we randomly chose 10 sources and 10 sinks, 33 sources and 33 sinks and finally 100 sources and 100 sinks of random security levels and analyzed the classified programs with our algorithm. We thereby measured the execution times of the whole algorithm, of the scan for probabilistic channels and of the scan for illicit and explicit flow. The test was run ten times and the presented results are the average values.

Table 1 shows the average execution times. 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 behavior is, besides the sheer size of the program, the number of sources and sinks. If only 10 sources and 10 sinks were selected, the size of the lattice in charge did not really matter. The runtime for lattice C was in several cases faster than that for lattice B or A. The cause of that behavior is that with 10 sources and 10 sinks the size of lattice C is not exhausted – the classification can introduce at most 20 different security levels to the analysis, the same maximal number as with lattice B. With 33 sources and 33 sinks the huge size of lattice C slowly became noticeable. Here the analysis for a program with lattice C was in most cases the most expensive. With 100 sources and 100 sinks the size of lattice C eventually became the dominating cost factor.

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. [11], which is employed for verifying the explicit and implicit flow. The results show that the two checks were similarly fast. However, it should be noted that the performance of the detection of probabilistic channels seems to decline stronger for huge lattices than Hammer et al.'s algorithm. For lattice C and 100x100 sources and sinks, the detection of probabilistic channels was in most cases more time-consuming. According to Hammer [9, 10], slicing-based IFC gets along with comparatively few annotations, which is an encouraging diagnosis.

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

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

# 6 Discussion and Related Work

In the following, we compare our variant of LSOD with probabilistic noninterference and LSOD definitions from the literature.

## 6.1 Weak Probabilistic Noninterference

Smith and Volpano's *weak probabilistic noninterference* (WPN) property [31, 35] 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. It is called 'weak' because the number of steps between two events in one run may differ from the number of steps between them in the other run.

WPN addresses explicit and implicit flow and probabilistic channels. Like in our analysis, timing channels

```
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;
```

Figure 10: Two examples comparing the restrictions of LSOD and weak probabilistic noninterference. 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.

and termination channels are excluded (which permits the probabilistic bisimulation to be weak). This renders their interpretation of low-observable behavior 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 between their definition of equivalent low-observable behavior and ours is that their definition disallows low-observable events to be delayed infinitely in the one low-observable behavior and being executed in the other. Thus, their definition is stricter with respect to termination channels and only permits the sheer termination of the program to differ.

**Attacker model.** The WPN attacker model is quite different from ours, and it is necessary to discuss the interdependence between attacker model and precision. WPN globally partitions the program variables into high and low, and the attacker is able to see all low variables at any time. Hence, the values of the low variables and their changes over time constitute the low-observable behavior. In contrast, our attacker can only see low operations and operands once they are executed, but cannot see all low variables at all times. 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. Theorem 1 guarantees soundness anyway.

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. Our mechanism aims to classify I/O operations as high or low and to treat unclassified operations as invisible, which in turn is not possible with Smith and Volpano's mechanism.

Why did we choose a weaker attacker? WPN pays a price for their stronger attacker model, in terms of precision and program restrictions. WPN is not flow sensitive which leads – in particular for concurrent programs – to strange false alarms. For example, the fragment `int h = PIN(); print(l); l = h; print(l2);` is considered unsafe by WPN, but safe by our LSOD. In fact flow-sensitive PDGs and slicing can only be applied under our attacker model, as under WPN all low variables must be considered, even those which are not found by slicing. Summarizing, WPN attackers see low memory, LSOD attackers see low events; it depends on the application context which attacker scenario is more realistic.

**Implementation.** Smith and Volpano present a security-type system that guarantees that well-typed programs are weakly probabilistic noninterferent. Expressions are classified as high or low, where an expression is low iff it does not contain any high variable. A program is weakly probabilistic noninterferent if

- only low expressions can be assigned to low variables,
- a conditional structure with a high guard cannot assign to a low variable and
- a conditional structure whose running time depends on high variables cannot be followed sequentially by assignments to low variables.

**Restrictions.** A weakness of Smith and Volpano's security-type system is that it lacks a detection of conflicts. Probabilistic channels are prevented by forbidding assignments to low variables sequentially behind conditional structures whose running time depends on high variables, which is very restrictive. The program on the left side of Fig. 10 is rejected by their security constraint, 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 our LSOD.

The type system assumes that a single statement has a fixed running time, which is not necessarily the case in reality. Based on this assumption, programs like that on the right side of Fig. 10 are accepted by their type system, 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 [31]. This is a disadvantage compared with LSOD, which holds for every scheduler. The type system does not support a modular verification of programs and libraries, which is also true for our LSOD checker.

## 6.2 Strong Security

Sabelfeld and Sands' security property *strong security* [30] 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 'lockstep execution' requirement allows to ignore the concrete scheduling strategy.

**Attacker model.** Programs are classified by partitioning variables into high and low variables. The attacker sees all low variables at any one time (see discussion above) and is aware of program termination, so the values of the low variables, their changes over time and the termination behavior constitute the low-observable behavior. 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 semantic effects on the low-observable behavior, which makes it possible to identify programs like that on the left side of Fig. 11 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 behavior is not, because both branches lead to 0 being assigned to `l`. Our algorithm is not able to recognize this program as secure and rejects it, the same holds for weak probabilistic noninterference.

Comparing their definition of low-observable behavior with ours and with the one of Smith and Volpano shows typical disagreement on the capabilities of the attacker. Sabelfeld/Sands and Smith/Volpano assume that the attacker is able to see low variables at any time, which we do not, 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/Sands.

**Advantages and restrictions.** 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 [29] 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, which is not compositional. On the other hand, lockstep execution imposes serious restrictions to programs, and an investigation of its practicability remains an important open issue.

The restriction to schedulers which do not touch high data means that any information possibly used by the scheduler, for example thread priorities or 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 makes it possible to break strong security by running the program under a scheduler for which the attacker knows the classification of the program to be inappropriate. This is a disadvantage compared with our technique, whose security property, secu-

```
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;
```

Figure 11: 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. 10, where the additional `skip` statement removes the probabilistic data channel.

rity constraint and classification mechanism are scheduler-independent.

**Implementation.** Sabelfeld and Sands present a security-type system which ensures that a well-typed program is strongly secure. The type system checks – similarly to that of Smith and Volpano – whether implicit and explicit flow is secure, and disallows loops with high guards completely in order to prevent termination channels.

Similar to Smith and Volpano, and in contrast to our LSOD, the authors assume that a single statement has a fixed execution time. Under that assumption, probabilistic channels may only appear if assignments to low variables are sequentially preceded by `if`-structures with high guards (since they forbid loops with high guards completely). The specific feature of their type system is that it transforms `if`-structures with high guards such that they cannot cause probabilistic channels. For that purpose, it pads the branches of such an `if`-structure with `skip`-statements until the branches have the same number of statements. For example, the program on the left-hand side of Fig. 10 would be transformed to the program on the right-hand side of Fig. 11 and would then be accepted.

## 6.3 LSOD by Zdancevic and Myers

Zdancevic and Myers [40] 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 address implicit and explicit flow as well as probabilistic data channels, called *internal timing channels* in [40]. They 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 probabilistic data channel in which the differing relative orders manifest.

A program is classified by partitioning variables into high and low, hence the low-observable behavior consists of the changes of low variables over time. Since the relative order of low-observable events does not matter, each low variable can be inspected in isolation. Let $T$ be a program run and $T(v)$ be the sequence of values a variable $v$ has during the program run, called the *location trace* of $v$. Two program runs $T$ and $U$ are low-equivalent if for every low variable $l$ the location traces $T(l)$ and $U(l)$ are equal up to the length of the shorter run and up to *stuttering*, which means that up to the shorter sequence $l$ undergoes the same changes in both program runs, but not necessarily at the same time. For example, if $T(l) = \langle 0,0,0,1,1,2,3\rangle$ and $U(l) = \langle 0,1,2\rangle$, then $T$ and $U$ are equivalent with respect to $l$.

The authors apply the approach to $\lambda_{SEC}^{PAR}$, a concurrent language with message-passing communication. Choosing message-passing makes the detection of data conflicts more specific: Data conflicts can only appear due to conflicting accesses to the same communication channel. The language provides *linear channels*, communication channels that are used for transmitting exactly one message and thus guarantee conflict-free communication. The authors present a security-type system 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 low-security observational deterministic if they are additionally free of data conflicts. However, a suitable analysis of data conflicts is not presented.

**Attacker model.** Zdancevic 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 in the host system. It is possible to modify our analysis to comply with their attacker, by simply skipping the detection of probabilistic order channels.

**Restrictions.** Their security constraint requires that programs are completely free of data conflicts, which is much stricter than ours. This requirement de facto prevents an application to languages with shared memory, because any program containing a data conflict would be rejected, even if the conflict does not influence the low-observable behavior at all.

## 6.4 LSOD by Huisman et al.

Huisman et al. [14] took up and improved the work of Zdancevic and Myers. They pointed out that Zdancevic and Myers' security property contains a leak, because its definition of low-equivalent program runs is restricted to the length of the shorter run. Consider the program on the left of Fig. 12, taken from [14], which copies a secret PIN to low variable `l`. It is sequential and therefore free of conflicts. Because of the loop, no two program runs with low-equivalent inputs and different input values for `h` have the same length. The additional assignments to `l` in the longer run, after which `h` has been copied to `l`, always fall out of the comparison. But up to the length of the shorter program run `l` has the same values in both program runs after every step, hence the program is accepted by Zdancevic and Myers' property. Huisman et al. [14] close that leak by strengthening the definition of low-equivalent program runs: Two program runs $T$ and $U$ are low-equivalent if for every low variable $l$ the location traces $T(l)$ and $U(l)$ are equal up to stuttering. This means that assignments to low variables sequentially behind loops iterating over high data are forbidden.

The authors present an additional definition of low-equivalent program runs that closes termination channels. It additionally requires that either both program runs terminate or none of them. They also describe the necessary measurements to encounter probabilistic order channels. This can be achieved 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 program runs up to stuttering.

```
void main():
  h = inputPIN();
  l = 0;
  while (h > 0)
    l++;
    h--;
```

```
void main():
  h = inputPIN();
  x = y;
  fork thread_1();
  fork thread_2();
void thread_1():
  x = 0;
  print(x);

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

Figure 12: Two examples demonstrating the strengths and weaknesses of Huisman et al.'s security property. Both programs are rejected, the first because it copies the PIN to the low variable `l`, the second because the order of the assignments `x = 0` and `y = 1` depend on interleaving.

The authors enforce their security property via model checking. They formalized their different security properties via two temporal logics, CTL$^*$ and the polyadic modal $\mu$-calculus, 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.

**Advantages and restrictions.** Huisman's security property is very flexible, as it permits to include and exclude termination channels and probabilistic order channels. But it is also more restrictive than ours. It is stricter towards termination channels, because 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 analyzed programs. Since a program is classified by partitioning variables into high and low, each assignment to a low variable is regarded as a low-observable event. The security property addressing probabilistic order channels requires that two low-equivalent program runs must make the same sequence of changes to low variables. This means in effect that if two threads work on different low variables, then the assignments to these variables must have a fixed interleaving order, even if the variables are completely unrelated (apart from being 'low').

As an example, consider the program on the right side of Fig. 12. Its main thread reads a PIN, assigns `y` to `x` and then forks both threads. Thread 1 sets `x` to 0 and then prints it, thread 2 sets y to 1. Assume that the PIN is high data and the output is low-observable. Using Huisman et al.'s technique, h is classified as a high variable and x as a low variable. Now it is compulsory that y is also classified as low because otherwise the assignment x = y would be illegal. This means that the assignments x = 0 and y = 1 are low-observable. Since the order of these assignments is not fixed, the program is rejected. Using our IFC technique, h = inputPIN() is classified as a high source and print(x) as a low sink, and the program is accepted by our security property. We therefore argue that our approach of classifying operations instead of variables is better suited for low-security observational determinism, because it permits a much less restrictive treatment of probabilistic order channels.

# 7 Future Work

## 7.1 Time sensitivity

Integration of a time-sensitive slicer instead of I2P will make our algorithm even more precise, but also much more expensive. Today, time-sensitive slicing is feasible for medium-sized programs, hence we plan to develop a time-sensitive LSOD checker. Algorithm engineering will be necessary to balance precision against scalability, and to use time-sensitivity only if necessary.

## 7.2 Lock sensitivity

As described in section 2, MHP analysis is crucial for precision of the PDGs and hence for overall IFC precision. The current MHP analysis however does not necessarily take into account explicit locks in the program. The latter property is called *lock sensitivity* and has been explored in [3, 4] in the scope of Dynamic Pushdown Networks. We recently integrated this analysis into our IFC. Preliminary experiments indicate that MHP indeed becomes more precise, as more interference edges are pruned.

## 7.3 Termination channels

Our security property excludes termination channels. This is common practice in IFC techniques for sequential programs, because termination channels are assumed to be sufficiently small, an assumption that does only hold for batch programs which can be seen as black boxes. As soon as programs interact with a user, termination channels can be used to leak an arbitrarily amount of information [1]. These channels can be prevented by forbidding low-observable behavior behind loops with guards that may receive high data, but this is a too severe restriction. A *termination analysis* for loops could solve that problem: Low-observable behavior behind such a loop can be permitted if its termination is guaranteeed by a static analysis. Several algorithms of varying precision for loop-termination analysis are available.

## 7.4 Declassification

We currently do not provide a declassification mechanism for probabilistic channels. Instead of declassifying probabilistic channels, we consider Zdancevic and Myers' idea of using *linear channels* for deterministic communication between threads [40] more promising. Linear channels can be integrated in form of a library into languages with shared memory. We have recently a added such a library as a proof-of-concept implementation, but our experiences with it are preliminary and are not reported here.

## 7.5 Machine-checked proofs

It is our long-term goal to formalize our LSOD 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 [38, 37].

## 7.6 Case studies

We will apply our analysis to reference scenarios in the DFG priority program "Reliably secure software systems"[5], which include an e-voting system and various Android apps. Analysis of the latter requires an adaption of our analysis to Android's Dalvik bytecode. Case studies will also allow to compare our LSOD criterion against other PN analysis methods with respect to precision, scalability, and usability.

[5] http://www.reliably-secure-software-systems.de/

# 8 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 of low-security observational determinism. It turns out that the criteria from the literature which are sufficient for LSOD can be naturally implemented through slicing algorithms for concurrent programs. We also demonstrated how our LSOD criterion fixes some weaknesses of earlier LSOD definitions from the literature.

Our implementation can handle full Java with an arbitrary number of threads. It was applied to several small examples from the literature; preliminary experience indicates high precision and scalability for medium-sized programs. It is planned to increase precision even more by using lock-sensitive MHP analysis algorithms.

Our current work is part of a long-standing project which aims to exploit modern program analysis for software security. We believe that precision, scalability, and usability of many security analyses can be greatly improved by applying recent achievements in program analysis algorithms. The current article demonstrates that IFC analysis of concurrent programs can indeed be improved by applying PDGs and MHP analysis; resulting in flow-sensitive, object-sensitive, context-sensitive, and time-sensitive algorithms.

**Acknowledgements.** We thank Andreas Lochbihler and Joachim Breitner for careful proofreading and discussions of this work.

# References

[1] 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.

[2] David Binkley, Mark Harman, and Jens Krinke. Empirical study of optimization techniques for massive slicing. *ACM Trans. Program. Lang. Syst.*, 30(1):3, 2007.

[3] 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.

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

[5] Dennis Giffhorn. Advanced chopping of sequential and concurrent programs. *Software Quality Journal*, 19(2):239–294, 2011.

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

[7] 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.

[8] Jürgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *Proc. 9th SCAM*, pages 105–114, September 2010.

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

[10] 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.

[11] 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.

[12] 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.

[13] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

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

[15] Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proc. ESEC/FSE-11*, pages 178–187, New York, NY, USA, 2003. ACM.

[16] Lin Li and Clark Verbrugge. A practical MHP information analysis for concurrent Java programs. In *Proc. 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, volume 3602 of *LNCS*, pages 194–208. Springer, 2004.

[17] 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.

[18] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

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

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

[21] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report 777, Computer Sciences Department, University of Wisconsin-Madison, 1988.

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

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

[29] 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.

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

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

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

[33] 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.

[34] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 30:27:1–27:30, September 2008.

[35] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999.

[36] 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.

[37] Daniel Wasserrab. Information flow noninterference via slicing. *Archive of Formal Proofs*, 2010, 2010.

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

[39] Bin Xin and Xiangyu Zhang. Efficient online detection of dynamic control dependence. In *Proc. ISSTA*, pages 185–195. ACM, 2007.

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

# Appendix A: Proof Sketch for Theorem 1

In the following, we describe the central steps in the proof of theorem 1. All details can be found in [6].

**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 6). $S(o,T)$ is thus similar to a dynamic backward slice for $o$. Similarly $D(o,T)$ is the *data 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 $q$ and $r$ be two operations which cannot happen in parallel, 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 $r$. Note that 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 of a program $p$ 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 resp. 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 $o \notin Pot(q)$, 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 3.2, 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 $\Theta$ 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.