On Time-Sensitive Control Dependencies

MARTIN HECKER, SIMON BISCHOF, and GREGOR SNELTING, Karlsruhe Institute of Technology

We present efficient algorithms for *time-sensitive* control dependencies (CDs). If statement y is time-sensitively control dependent on statement x, x not only decides whether y is executed, but also how many time steps after x. If y is not standard control dependent on x, but time-sensitively control dependent, then y will always be executed after x, but the execution time between x and y varies. This allows to discover e.g. timing leaks in security-critical software.

We systematically develop properties and algorithms for time-sensitive CDs, as well as for nontermination-sensitive CDs. These do not only work for standard control flow graphs (CFGs), but also for CFGs lacking a unique exit node (e.g. reactive systems). We show that Cytron's efficient algorithm for dominance frontiers [10] can be generalized to allow efficient computation not just of classical CDs, but also of time-sensitive and nontermination-sensitive CDs. We then use time-sensitive CDs and time-sensitive slicing to discover cache timing leaks in an AES implementation. Performance measurements demonstrate scalability of the approach.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → **Program analysis**; • **Security and privacy** → **Logic and verification**.

Additional Key Words and Phrases: control dependency, program slicing, timing dependency, timing leak

ACM Reference Format:
Martin Hecker, Simon Bischof, and Gregor Snelting. 2021. On Time-Sensitive Control Dependencies. *ACM Trans. Program. Lang. Syst.* 1, 1 (August 2021), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND OVERVIEW

Timing Leaks are a major source of software security problems today. Attacks based on timing leaks such as *Spectre* [22] have become known to the general public. Yet there are not many program analysis tools that detect timing leaks in software.

In this article we describe a new kind of dependency between program statements, the *time-sensitive control dependency*. It is able to discover timing leaks, and can be implemented as an automatic program analysis. We will explain time-sensitive dependencies, provide efficient algorithms, provide a soundness proof, and apply it to discover timing leaks in an implementation of the AES cryptographic standard.

The construction of time-sensitive control dependencies starts with classical control dependencies. We will thus begin by sketching the research path from CDs to timing

Authors' address: Martin Hecker; Simon Bischof; Gregor SneltingKarlsruhe Institute of Technology, gregor.snelting@kit.edu.

0164-0925/2021/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

dependencies, and provide introductory examples. Later in the article, we will provide formal definitions, proofs, and algorithms.

Control dependencies (CDs), originally introduced by [11, 33], are a fundamental building block in program analysis. CDs have many applications: they can, for example, be used for program optimizations such as code scheduling, loop fusion, or code motion (see e.g. [25]); or for program transformations such as partial evaluation (e.g. [20]) or refactoring (e.g. [5]). CDs are in particular fundamental for program dependence graphs (PDGs) and program slicing [11, 19, 23]. Intuitively, a program statement y is control dependent on another statement x, written $x \to_{cd} y$, if x – typically an **`if`** or **`while`** statement – decides whether y will be executed or not. Classical CDs are defined via postdominators; in fact classical CDs are essentially the postdominance frontiers in the control flow graph (CFG) [10]. Postdominance frontiers can be computed by an efficient algorithm due to Cytron [10].

Unfortunately, the classic CD definition is limited to CFGs with unique exit node, and thus assumes that all programs can terminate. In 2007 Ranganath et al. [29] generalized control dependence to CFGs without unique exits and nonterminating programs (e.g. reactive systems); providing the first algorithm for nontermination-sensitive CDs. Later, Amtoft [3] provided definitions and algorithms for nontermination-insensitive CDs, which allow for sound analysis and slicing of nonterminating programs. But these algorithms could no longer be based on the efficient Cytron algorithm for postdominance frontiers.

In this contribution, we not only present new efficient algorithms for the Ranganath-Amtoft CD definitions. We will also provide definitions and algorithms for *time-sensitive* CDs and time-sensitive slicing. Time-sensitive CD, written $x \to_{\text{tscd}} y$, holds if x decides whether y is executed, or if x decides *when* y is executed (even if y is always executed after x) – that is, how many time units after execution of x. Intuitively, $x \to_{\text{tscd}} y$ while *not* $x \to_{cd} y$ means that y will always be executed after x, but the execution time between x and y varies. The latter property is important to discover timing leaks in security-critical software.

We systematically develop theoretical properties and efficient algorithms for $\to_{\text{tscd}}$, and evaluate their performance. It turns out that Cytron's efficient algorithm for dominance frontiers can be generalized to an abstract notion of dominance, which then can be used for the efficient computation of both the Ranganath/Amtoft CD, as well as our new time-sensitive CD. We then apply $\to_{\text{tscd}}$ to (models of) hardware microarchitectures, and use it to find cache timing leaks in an AES implementation.

Many of the theorems in this article have been formalized and machine-checked using the machine prover *Isabelle*. Such theorems are marked with a sign. The Isabelle proofs can be found in the electronic appendix of this article. For some theorems in section 5, such an Isabelle proof has not yet been completed. Manual proofs are available, but are not presented in this article. Consequently, such theorems are called "observations".

1.1 Overview

The main part of this article will present time-sensitive CDs and algorithms in a rather technical manner. Before we embark on this, we present an informal overview of our research path and results. We begin with a discussion of classical control dependencies, and compare these to our new notion of time-sensitive control dependencies.

1.1.1 Control Dependence. Informally, a control dependence in a CFG, written $x \to_{cd} y$, means that x decides whether y is executed or not. In structured programs, x is typically an **`if`** or **`while`** statement. Figure 1 presents two examples: In the first example (left), node (5) is control dependent on node (1); node (3) is not control dependent on (1) but on

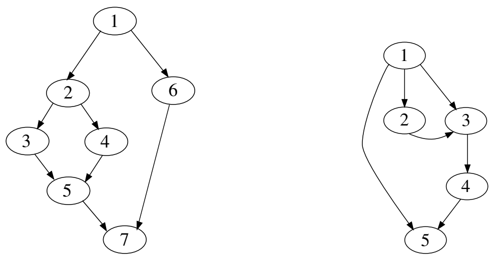


Fig. 1. Two simple control flow graphs illustrating control dependence

(2). In the second example (right)[1], nodes (2), (3), and (4) are control dependent on node (1). Technically, CD is based on the notion of *postdomination* in CFGs. y postdominates x (written $y \sqsubseteq_{\text{POST}} x$) if any path from x to the *exit* node must pass through y. Several formal CD definitions exist; as this may be confusing we will relate the most popular definitions to the examples in Figure 1. The original definition of CD in [11] is as follows:

$$x \rightarrow_{cd} y \iff \neg\big(y \sqsubseteq_{\text{POST}} x\big) \land \exists \text{ path } \pi : x \rightarrow^* y \text{ such that } \forall z \in \pi \setminus \{x, y\} : y \sqsubseteq_{\text{POST}} z$$

The condition that y is not a postdominator for x means that from x there is a second path (not containing y) to the exit node. That is, there is a conditional branch at x. The next condition demands that there is a path from x to y, and that y is a postdominator for all nodes z between x and y. Thus there is no side branch from any z to the exit node; hence x is directly controlling whether y is executed.

In Figure 1 (left), node (5) postdominates all nodes on paths between (1) and (5), but (5) does not postdominate (1); hence $(1) \rightarrow_{cd} (5)$. But (3) does not postdominate (2) (this node being the only one between (1) and (3)), hence $\neg\big((1) \rightarrow_{cd} (3)\big)$. In Figure 1 (right), node (4) is control dependent on node (1). Since we have $(1) \rightarrow (5)$, node (4) does not postdominate (1). The path $(1) \rightarrow (3) \rightarrow (4)$ only contains the additional $z = (3)$, and (4) postdominates (3), so the second condition is satisfied. But what about the path $(1) \rightarrow (2) \rightarrow (3) \rightarrow (4)$? It is irrelevant, as the CD definition only demands there *exists* a path where for all z etc; it does not demand the z condition for all paths. Likewise, (2) as well as (3) are control dependent on (1).

An alternate, more compact CD definition was provided in [33], and is used in this article. Here x is a branch node with direct successors n and m, where the control-dependent y postdominates one but not the other:

$$x \rightarrow_{cd} y \iff \exists n, m : x \rightarrow n, x \rightarrow m, y \sqsubseteq_{\text{POST}} n, \neg\big(y \sqsubseteq_{\text{POST}} m\big)$$

Lemma 1.1. The above definitions for $x \rightarrow_{cd} y$ are equivalent whenever $x \neq y$.

Applied to Figure 1 (right), again we conclude that (4) is CD on (1). Choose $n = (2)$ ($n = (3)$ also works), $m = (5)$, then (4) postdominates (2) (and (3)), but (4) does not postdominate (5). Note that both n and m in the definition are existentially quantified. Thus the definition neither demands nor inhibits that (4) postdominates (3).

Lemma 1.2. In Figure 1 (right), we have $(4) \sqsubseteq_{\text{POST}} (2)$, $(4) \sqsubseteq_{\text{POST}} (3)$, $\neg\big((4) \sqsubseteq_{\text{POST}} (5)\big)$, and thus $(1) \rightarrow_{cd} (4)$.

[1]We thank one reviewer for suggesting this example.

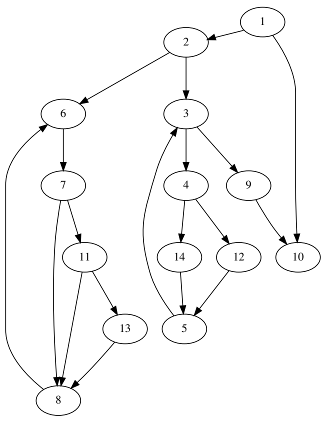


Fig. 2. A CFG G

1.1.2 CFGs Without Unique Exit. CFGs without unique exit, in particular with no exit or unreachable exits, are important for modern language constructs, for example event handlers or loops in reactive systems. Ranganath and Amtoft had generalized postdominance and CD for such CFGs. The resulting postdominance relations are called max- and sink-postdominance, and will be explained in section 2.2. If these are used in CD definitions, one obtains *non-termination-sensitive* control dependence, written $\rightarrow_{\text{ntscd}}$; and *non-termination-insensitive* control dependence, written $\rightarrow_{\text{nticd}}$. $\rightarrow_{\text{nticd}}$ is identical to $\rightarrow_{cd}$, but also works for graphs without unique exit. $\rightarrow_{\text{ntscd}}$ is identical to $\rightarrow_{wcd}$ (weak control dependence, see section 2.1), but also works for graphs without unique exit.

$\rightarrow_{\text{nticd}}$ and $\rightarrow_{\text{ntscd}}$ are important building blocks for $\rightarrow_{\text{tscd}}$. A comparison between $\rightarrow_{\text{nticd}}$, $\rightarrow_{\text{ntscd}}$, and $\rightarrow_{\text{tscd}}$ is given in the next section.

1.1.3 Time-Sensitive Control Dependence. Time-sensitive CD, written $x \rightarrow_{\text{tscd}} y$, holds if x decides *when or whether* y is executed. This dependence is more relaxed than standard CD. Intuitively, $x \rightarrow_{\text{tscd}} y$ while *not* $x \rightarrow_{cd} y$ means that y will always be executed after x, but *the execution time between x and y varies.*

The latter property is important to discover timing leaks in security-critical software. A typical situation is as follows: y is not control dependent on x, but there are at least two paths from x to y. Then the run time between x and y varies: $x \rightarrow_{\text{tscd}} y$. If this variation depends on secret data, and can be measured by an attacker, a timing leak has been born. $x \rightarrow_{\text{tscd}} y$ will uncover this leak.

In our work time is discrete; a unit of time coincides with a transition in the CFG. Since steps of an abstraction of real programs and hardware are timed, this is therefore a “weakly timing-sensitive” model in the sense of [21].

Now, let us illustrate the differences between the different kinds of control dependences. A node y is non-termination sensitively control dependent on node x, written $x \rightarrow_{\text{ntscd}} y$, if x decides whether y will be executed. In Figure 2, we have $(1) \rightarrow_{\text{ntscd}} (2)$, because we will execute (2) when choosing (2) as the successor of (1) but not if we choose (10). Also, due to the loop at (3), we have $(3) \rightarrow_{\text{ntscd}} (10)$: By choosing (9) as the successor of (3) we are guaranteed to reach (10). But if we choose (4) as the successor, we might avoid reaching (10) by staying in the loop forever, so (3) decides if (10) will be executed.

y is non-termination insensitively control dependent on x, written $x \to_{\text{nticd}} y$, if x decides whether y will be executed, *assuming we eventually exit all loops that can be exited.* In Figure 2, we still have $\textcircled{1} \to_{\text{nticd}} \textcircled{2}$, with the same reasoning as above. But now we have $\neg\left(\textcircled{3} \to_{\text{ntscd}} \textcircled{10}\right)$: Since we assume that we always exit the loop at $\textcircled{3}$, we are guaranteed to reach $\textcircled{10}$, no matter which successor we choose at $\textcircled{3}$.

y is timing sensitively control dependent on x, written $x \to_{\text{tscd}} y$, if x decides *when* y will be executed. In Figure 2, we have $\textcircled{7} \to_{\text{tscd}} \textcircled{8}$, because we will execute $\textcircled{8}$ after one step when choosing $\textcircled{8}$ as the successor of $\textcircled{7}$ but not if we choose $\textcircled{11}$, when it takes two or three steps. Therefore, the choice taken at $\textcircled{7}$ influences the timing of $\textcircled{8}$. On the contrary, $\neg\left(\textcircled{4} \to_{\text{tscd}} \textcircled{5}\right)$, because no matter how we choose, we will always reach $\textcircled{5}$ in two steps. An interesting case is $\textcircled{1} \to_{\text{tscd}} \textcircled{2}$: If we choose $\textcircled{2}$ as successor of $\textcircled{1}$, we will reach $\textcircled{2}$ in exactly one step, but we will not if we choose $\textcircled{10}$ because we then will not reach $\textcircled{2}$.

1.1.4 Applications for Software Security. As indicated, $\to_{\text{tscd}}$ may help to discover timing leaks. More generally, $\to_{\text{tscd}}$ is useful for *Information Flow Control* (IFC). IFC uses program analysis techniques to discover leaks in software. Technically, *noninterference* is a property which guarantees that a program does not leak secret data. *Probabilistic* noninterference guarantees that there are no internal timing leaks, which arise if secret data influence scheduling or other measurable timing properties. For an introduction to IFC, see e.g. [31].

Indeed $\to_{\text{tscd}}$ was developed as an instrument to improve the precision of probabilistic noninterference analysis. We will report on applications of $\to_{\text{tscd}}$ for IFC in a separate article. In the current article, we focus on algorithms for $\to_{\text{tscd}}$; and use a different security example: in section 4, we will analyse an implementation of the AES cryptographic standard, and discover cache leaks in this implementation. These infamous cache leaks have been known for some time [4], but so far no program analysis tool was able to discover such leaks.

1.1.5 Algorithms. The major part of this contribution is concerned with efficient algorithms for $\to_{\text{tscd}}$. For the classical $\to_{cd}$, Cytron's efficient algorithm for dominance frontiers can be used; but this algorithm was not employed by Ranganath/Amtoft.

We discovered that a generalized version of Cytron's algorithm can not only be used for both $\to_{\text{nticd}}$ and $\to_{\text{ntscd}}$, but also for $\to_{\text{tscd}}$. Thus we have been able to obtain efficient implementations for all these dependence notions. The algorithms are described in section 5. Performance evaluations are described in section 6.

2 CONTROL DEPENDENCE IN GRAPHS WITHOUT UNIQUE EXIT

Our work was strongly motivated by earlier results of Ranganath et al. [29] and Amtoft [3]. These authors extended the classical notion of CD and slicing to CFGs which do not contain a unique exit node. As multiple exit nodes can trivially be handled by adding a new "global" exit node, Ranganath's and Amtoft's work is in fact concerned with CFGs which do not have a single, unique exit node. A typical example is a CFG with an infinite loop from which an exit node cannot be reached. Such CFGs are relevant, because modern programs need not necessarily terminate through exit nodes. One paramount example are reactive systems, which are assumed to run forever; and thus have no exit node at all. Another example are event handlers, which may shutdown a thread while the thread has no explicit exit. Thus Ranganath and Amtoft opened the door to apply CD and slicing to modern program structures.

Time-sensitive CD will also work on graphs without unique exit. It is therefore necessary to recall Ranganath's and Amtoft's work. We begin with fundamental definitions of CDs and postdomination for CFGs with no unique exit.

2.1 Classical Control Dependence and Weak Control Dependence

CFGs are a standard representation of programs e.g. in compilers, and many tools are available which extract CFGs from source code.[2] Thus let $G = (N, \to_G)$ be the CFG of a program. In this article, we once and for all assume a fixed CFG G and therefore omit the sub- or superscript G whenever possible; e.g. we write $n \to m$ instead of $n \to_G m$. In the classical case of a unique exit node, there is $\mathit{exit} \in N$ such that $n \to^* \mathit{exit}$ for all $n \in N$, and $\mathit{exit} \to n$ for *no* node $n \in N$.

Node m postdominates n ($m \sqsubseteq_{\text{POST}} n$) iff $m \in \pi$ for every path π from n to exit. Node m *strongly* postdominates n ($m \sqsubseteq_{\text{SPOST}} n$) iff $m \sqsubseteq_{\text{POST}} n$, *and* there exists some $k \geq 1$ such that $m \in \pi$ for every path π starting in n with length $\geq k$ [27]. In contrast to $m \sqsubseteq_{\text{POST}} n$, $m \sqsubseteq_{\text{SPOST}} n$ does *not* hold if there is an infinite loop between m and n: Assume such a loop exists, then there will be paths π starting at n of arbitrary length k which never pass through m: $\forall k \exists \pi : \text{len}(\pi) = k \wedge m \notin \pi$. If this happens, $\sqsubseteq_{\text{SPOST}}$ is *not* supposed to hold; hence the negation of the latter condition must hold for $\sqsubseteq_{\text{SPOST}}$.

Classical (nontermination-insensitive) CD, denoted $\to_{cd}$, is defined in terms of postdominance. Formally (as already explained above),

$$x \to_{cd} y \iff \exists n, m \in N : x \to n, x \to m, y \sqsubseteq_{\text{POST}} n, \neg(y \sqsubseteq_{\text{POST}} m)$$

This CD definition can be modified in order to react sensitively to infinite loops. This *nontermination sensitive* form of CD, called "weak control dependence" and written $x \to_{wcd} y$, was introduced in [27]; and is defined in terms of *strong* postdominance. The formal definition is identical to the above CD definition; with $\sqsubseteq_{\text{SPOST}}$ instead of $\sqsubseteq_{\text{POST}}$. Even if $x \to_{cd} y$ does not hold, $x \to_{wcd} y$ might still hold if there is an infinite loop between x and y. Note that weak control dependence does not imply that this infinite loop is in fact executed.

2.2 Postdominance in Graphs Without Unique Exit

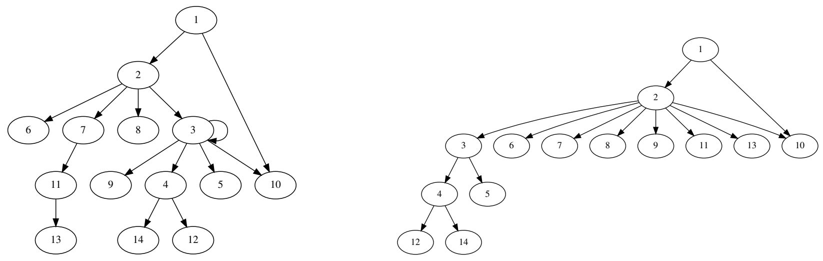
In order to understand how the above definitions are generalized to arbitrary graphs with no unique exit node, consider the example in Figure 2. It has no unique exit node, since the only candidate node 10 is unreachable from, e.g., node 6. Thus the classical definitions for $\sqsubseteq_{\text{POST}}$ and $\sqsubseteq_{\text{SPOST}}$ cannot be applied. Instead in [29], Ranganath et al. proposed control dependence for arbitrary graphs based on the notions of *maximal* and *sink* paths.

A maximal path is a path which cannot be extended (i.e.: is infinite, or ends in some node n without successor). On the other hand, a (control-) sink is a strongly connected component S such that no edge leaves S.[3] Specifically, all nodes n without successor (in particular $n = \mathit{exit}$) form a (trivial) sink. A sink path then is a path π such that $s \in S$ for some node $s \in \pi$ and some sink S, and if S is nontrivial (i.e. not a singleton), then all nodes in S appear in π infinitely often. In programming terms, S would be an infinite loop in the CFG, and a sink path corresponds to an execution which infinitely loops in S.

Definition 2.1 (Implicit in [29]). A node $m \in N$ nontermination-sensitively postdominates a node $n \in N$ (written $m \sqsubseteq_{\text{MAX}} n$) iff $m \in \pi$ for all maximal paths π starting in n. Similarly, a

[2] All examples and measurements in this article are based on CFGs which were produced using the JOANA system. JOANA is a system for IFC and can in particular check probabilistic noninterference for full Java with arbitrary threads [6, 13, 14].

[3] In a strongly connected component (SCC) S, there is a path between all $x, y \in S$. Every cycle is an SCC.



(a) $\to_{\text{ntscd}}$ (b) $\to_{\text{nticd}}$

Fig. 3. Nontermination-(in)sensitive CDs for CFG from Figure 2

node m nontermination-insensitively postdominates a node n (written $m \sqsubseteq_{\text{SINK}} n$) iff $m \in \pi$ for all sink paths π starting in n.

Since every sink path is a maximal path, $m \sqsubseteq_{\text{MAX}} n$ implies $m \sqsubseteq_{\text{SINK}} n$. $m \sqsubseteq_{\text{SINK}} n$ while $\neg\left(m \sqsubseteq_{\text{MAX}} n\right)$ means that on reaching n, m will later be executed unless an infinite loop is entered.

The following definition is equivalent to those in [29] whenever $n \neq m$.

Definition 2.2. A node $y \in N$ is *non-termination sensitively (resp. insensitively) control-dependent on* $x \in N$, written $x \to_{\text{ntscd}} y$ (resp. $x \to_{\text{nticd}} y$), if there exist edges $x \to n$, $x \to m$ such that $y \sqsubseteq_{\text{MAX}} n$ (resp. $y \sqsubseteq_{\text{SINK}} n$), but $\neg\left(y \sqsubseteq_{\text{MAX}} m\right)$ (resp. $\neg\left(y \sqsubseteq_{\text{SINK}} m\right)$).

Note that this definition is identical to the original CD definition, with $\sqsubseteq_{\text{MAX}}$ resp. $\sqsubseteq_{\text{SINK}}$ instead of $\sqsubseteq_{\text{POST}}$. In fact, for graphs with unique exit node, we have $\to_{\text{ntscd}} = \to_{wcd}$ and $\to_{\text{nticd}} = \to_{cd}$ [29]. Figure 3 shows $\to_{\text{ntscd}}$ and $\to_{\text{nticd}}$ for the CFG from Figure 2.

Like many program analysis problems, postdominance and CD can be characterized as a fixpoint computation. Our first new insight is that both $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ can be characterized as a greatest resp. least fix point of *one* rule set D. This surprising fact is the basis for our generalization of Cytron's algorithm. Note that the rule set can be interpreted as a functional which transforms a set of dominance relationships $\{x \sqsubseteq y\}$ into a new set $\mathsf{D}\left(\{(x \sqsubseteq y\}\right)$. If such a functional is monotone, it has a least as well as a greatest fixpoint.

Theorem 2.1. [4] Let D be the following rule system, and let D also denote its implicit functional; write μ for the least fixpoint, and ν the greatest fixpoint. Then D is a monotone functional in the (finite) lattice $\left(2^{N\times N}, \subseteq\right)$, and $\mu\mathsf{D} = \sqsubseteq_{\text{MAX}}$, and $\nu\mathsf{D} = \sqsubseteq_{\text{SINK}}$.

$$\text{Rule system } \mathsf{D}: \qquad \frac{}{n \sqsubseteq n}\,\mathsf{D}^{\text{self}} \qquad \frac{\forall n \to x:\ m \sqsubseteq x \qquad n \to^{*} m}{m \sqsubseteq n}\,\mathsf{D}^{\text{succ}}$$

The reachability side-condition $n \to^{*} m$ is in most cases redundant for the least fixed point $\mu\mathsf{D}$, but essential for the greatest fixed point $\nu\mathsf{D}$.[5]

Of course, algorithms for $\to_{\text{ntscd}}$ and $\to_{\text{nticd}}$ are needed, and indeed Ranganath et al proposed such algorithms. The algorithm for $\to_{\text{ntscd}}$ from [29] can in principle be thought of

[4]Lemmas and Theorems marked with have been formalized and proved in the machine prover Isabelle. The proof explanations and scripts can be found in the electronic appendix of this article.
[5]We mention in passing that for graphs with unique exit node, replacing this condition with $n \neq exit$ results in a similar rule system P on which the algorithm from [8] is based. We will not describe P in detail, but note that $\sqsubseteq_{\text{POST}} = \nu\mathsf{P}$ [15].

as a simple least fixed point computation of the set of nodes m such that $m \sqsubseteq_{\text{MAX}} n$, but only for nodes n that are successors of *branching* nodes.

We however discovered a more general and systematic algorithmic approach, which exploits the above fix-point theorem. It is based on the insight that Cytron's efficient algorithm for dominance frontiers can be generalized to an abstract notion of "dominance"; and thus can be used for $\sqsubseteq_{\text{POST}}$, $\sqsubseteq_{\text{SPOST}}$, $\sqsubseteq_{\text{MAX}}$, $\sqsubseteq_{\text{SINK}}$, $\rightarrow_{\text{ntscd}}$, $\rightarrow_{\text{nticd}}$, and in particular for $\rightarrow_{\text{tscd}}$. We will present all algorithms in a separate section (section 5), as they demand a rather heavy technical machinery.

In conclusion of this section, we recall another notion from Ranganath et al., which will also be helpful to characterise $\rightarrow_{\text{tscd}}$.

Definition 2.3 ([29]). *Decisive order dependence*, written $n \rightarrow_{\text{dod}} (m_1, m_2)$ is a ternary relation which means that n controls the *order* in which m_1, m_2 are executed.

We omit the formal definition, but provide an intuition: In [29], the necessity of $\rightarrow_{\text{dod}}$ was motivated by an irreducible[6] graph, such as the graph shown in Figure 6a. Here, neither m_1 nor m_2 is nontermination sensitively control dependent on n: $\neg(n \rightarrow_{\text{ntscd}} m_1) \wedge \neg(n \rightarrow_{\text{ntscd}} m_2)$. But the decision at n determines which node is executed next: Leaving n via the left branch will execute m_1 before m_2, but leaving n via the right branch will execute m_2 before m_1. Thus $n \rightarrow_{\text{dod}} (m_1, m_2)$ holds. Ranganath and Amtoft used $\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{dod}}$ to define a sound notion of *nontermination-sensitive backward slicing*. This is consistent with the fact that CDs are fundamental for slicing and PDGs.

3 TIMING SENSITIVE CONTROL DEPENDENCE

3.1 Why Time-Sensitivity Matters

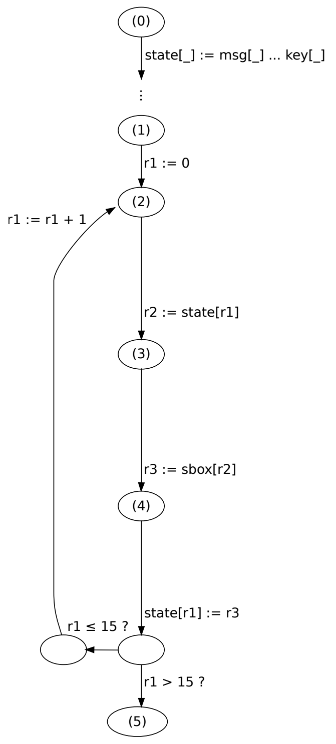
Before we formally develop timing sensitive CDs, let us motivate the usefulness of this concept for software security analysis. Known attacks exploiting timing side channels include Spectre [22] and cache attacks on implementations of the cryptographic standard AES [4]. In this kind of attacks, the attacker is able to observe the timing behaviour of certain instructions; from this observation determine whether some specific data are in the cache or not; and from this knowledge infer values of secret variables (e.g. by using the secret value as an array index), or draw conclusions about control flow.

Timing sensitive CDs can reveal such potential attacks, or prove that such attacks are impossible. For example, in a specific AES implementation[7] we find the code lines

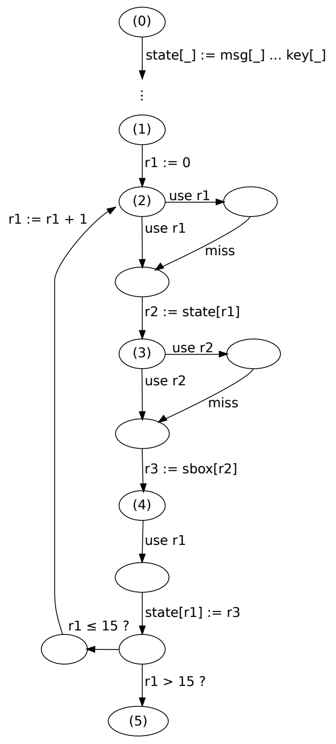
```
(1)   for r1: [0,1,...,15]
(2)      r2 := state[r1];// state depends on key and plain text
(3)      r3 := sbox[r2]; // sbox is a constant array
(4)      state[r1] := r3
(5)   end
```

`sbox` is a constant array which typically spans multiple memory blocks, while `r1`, `r2`, `r3` are registers. Thus the value of `r2` in one iteration may influence whether the read of `r3` in a later iteration is served from cache (namely if, earlier, the corresponding memory block was already loaded into the cache), or from main memory. This makes a difference in execution time and can be observed by an attacker; who may thus be able to infer the value of `r2`.

[6]A CFG is reducible if the forward edges form a directed acyclic graph, and in any backedge $m \rightarrow n$, n dominates m. Structured programs have reducible CFGs; wild `goto`s typically lead to irreducible CFGs.
[7]This implementation was presented in [4]. It assumes that all accesses to the `sbox` array need constant time. But in fact access time is cache dependent.



(a) Standard CFG



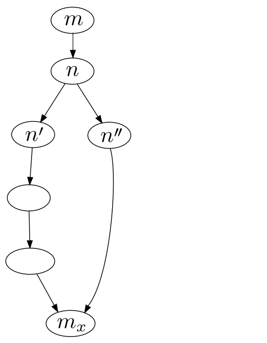
(b) Cache Aware CFG

Fig. 4. Control Flow Graphs for AES Sbox substitution.

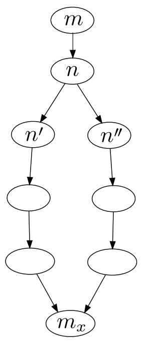
Such leaks can be discovered by timing-sensitive CDs; provided the CFG not just describes the code, but additionally models relevant hardware features such as cache behaviour.

Figure 4 shows the standard CFG for the AES code fragment, as well as the *micro-architectural CFG* which models timing differences due to cache hits and misses. The latter CFG indicates that array access `r3 := sbox[r2]` in line (3) may either result in a cache hit or cache miss. In the control flow, this is modeled via two paths leaving node (3) that are joined after following a *different* number of edges, and take a different amount of time to execute. Specifically, the time at which execution reaches the exit node (5) depends on which paths are taken at (3): node (5) is time sensitively control dependent on node (3). The edge annotations `use r2` indicate that the value register `r2` determines which array index, and hence which cache line is accessed at (3). Furthermore, due to the previous assignment `r2 := state[r1]`, node (3) is data-dependent on the initialization of the state array from the plain text message and the key, as indicated in node (0).[8] Thus we obtain the following dependency chain (where $\to_{dd}$ denotes data dependency): (secret input) $\to_{dd}$ `state` $\to_{dd}$ `r2` $\to_{dd}$ (3) $\to_{\text{tscd}}$ (5). That means: the time until (5) is reached depends on secret input. Thus time-sensitive CDs reveal that `sbox` access is not constant time (in contrast to the AES specification); opening a door to cache-leak based attacks.

[8]Besides CDs, data dependencies are important for security analysis. This is described in section 3.4. For the current AES example, the reader may assume all data dependencies are available as necessary.



(a) A CFG with external timing leak



(b) A CFG without timing leak

Fig. 5. Dependence of execution time of m_x on n.

If a $\to_{\text{tscd}}$ dependency is not (indirectly) data dependent on secret data, it does *not* generate a timing leak. E.g. in the AES CFG, (2) $\to_{\text{tscd}}$ (5) also holds. But the value of `r1` (and thus (2)) is not data dependent on any secret value, so no timing leak arises via (2). We will discuss this in more depth in section 3.4; and come back to AES and micro-architectural CFGs in section 4.

3.2 Timing Sensitive Control Dependence

Consider Figure 5a: m_x is guaranteed to be executed, no matter which branch is taken at n, so we have $\neg\left(n \to_{\text{ntscd}} m_x\right)$. But let us assume that we could measure execution times. Now, n can control at which time m_x will be executed, namely 4 or 2 steps after executing n. We will say that m_x is timing sensitively control-dependent on n, or $n \to_{\text{tscd}} m_x$. In Figure 5b, however, m_x will always be executed 4 steps after n, so there we have $\neg\left(n \to_{\text{tscd}} m_x\right)$.

We will now formally define $\to_{\text{tscd}}$. Specifically, we will

(1) Propose a notion $\sqsubseteq_{\text{TIME}}$ of *timing sensitive* postdominance.
(2) Give a least fixed point characterization of $\sqsubseteq_{\text{TIME}}$.
(3) Propose a notion $\to_{\text{tscd}}$ of *timing sensitive* control dependence. It will be based on $\sqsubseteq_{\text{TIME}}$ the same way that $\to$ntscd is based on $\sqsubseteq_{\text{MAX}}$.
(4) Prove soundness and minimality of $\to_{\text{tscd}}$.

To start with, remember that $\sqsubseteq_{\text{MAX}}$ was defined via

$$m \sqsubseteq_{\text{MAX}} n \iff \forall \pi \in {}_n\Pi_{\text{MAX}}.\;\; m \in \pi$$

where ${}_n\Pi_{\text{MAX}}$ is the set of maximal paths starting in n. For example, in Figure 5a it holds that $m_x \sqsubseteq_{\text{MAX}} n$, because any maximal path starting in any successor of n must contain m_x (i.e.: both $m_x \sqsubseteq_{\text{MAX}} n'$ and $m_x \sqsubseteq_{\text{MAX}} n''$), and so must any maximal path starting in n.

Now for time-sensitive postdominance we additionally want to express that in Figure 5a m_x can be reached via two different paths, *with varying execution time*. In order to account for the different *timing* of the (first) occurrence of m_x in maximal paths starting in n, the following auxiliary definition is needed.

Definition 3.1. Given any path $\pi = m_0, m_1, m_2, \ldots$ we say that m appears in π at position k iff $m = m_k$, and write $m \in^k \pi$. If additionally, $m_i \neq m$ for all $i < k$, we say that m *first* appears in π at position k, and write $m \in^k_{\text{FIRST}} \pi$.

Using this notation, we can define time-sensitive postdominance as follows.

Definition 3.2. (a) m timing-sensitively postdominates n at position $k \in \mathbb{N}$, written $m \sqsubseteq^{k}_{\text{TIME}} n$, iff on all maximal paths starting in n, m first appears at position k. Formally

$$m \sqsubseteq^{k}_{\text{TIME}} n \iff \forall \pi \in {}_{n}\Pi_{\text{MAX}}.\ m \in^{k}_{\text{FIRST}} \pi$$

(b) m timing-sensitively postdominates n, written $m \sqsubseteq_{\text{TIME}} n$, if there exists k such that $m \sqsubseteq^{k}_{\text{TIME}} n$. Thus

$$m \sqsubseteq_{\text{TIME}} n \iff \exists k \in \mathbb{N}\ \forall \pi \in {}_{n}\Pi_{\text{MAX}}.\ m \in^{k}_{\text{FIRST}} \pi$$

If we compare $m \sqsubseteq^{k}_{\text{TIME}} n$ to $m \sqsubseteq_{\text{MAX}} n$, the difference is that in the latter, m must occur somewhere in all maximal paths from n, while in the former m must first occur at a specific position k in all maximal paths from n. Thus if $m \sqsubseteq_{\text{TIME}} n$, m must appear in all maximal paths from n *at the same position*. Therefore in Figure 5a, $m_x \sqsubseteq_{\text{TIME}} n$ does not hold, while in Figure 5b, $m_x \sqsubseteq_{\text{TIME}} n$ does hold.

Lemma 3.1. Given m and n, the k such that $m \sqsubseteq^{k}_{\text{TIME}} n$ (if it exists) is unique.

Following the definitions for nontermination sensitive and insensitive control dependence $\to_{\text{ntscd}}$ and $\to_{\text{nticd}}$, we define the following timing sensitive notion of control dependence:

Definition 3.3. y is said to be *timing sensitively control-dependent* on x, written $x \to_{\text{tscd}} y$, if there exist edges $x \to n$ and $x \to m$ as well as some $k \in \mathbb{N}$ such that

$$y \sqsubseteq^{k}_{\text{TIME}} n \text{ and } \neg(y \sqsubseteq^{k}_{\text{TIME}} m)$$

Note that this definition is identical to the definition of $\to_{\text{ntscd}}$ resp. $\to_{cd}$; with $\sqsubseteq^{k}_{\text{TIME}}$ instead of $\sqsubseteq_{\text{MAX}}$ resp. $\sqsubseteq_{\text{POST}}$. Thus $\to_{\text{tscd}}$ has the same formal structure as classical CD and its later extensions.

In Figure 5a we have $m_x \sqsubseteq^{3}_{\text{TIME}} n'$ but $\neg(m_x \sqsubseteq^{3}_{\text{TIME}} n'')$, thus we have $n \to_{\text{tscd}} m_x$; while in Figure 5b we have $m_x \sqsubseteq^{3}_{\text{TIME}} n'$ and $m_x \sqsubseteq^{3}_{\text{TIME}} n''$ and thus *not* $n \to_{\text{tscd}} m_x$. For more complex examples, consider again the CFG in Figure 2. The timing sensitive postdominance for this CFG is shown in Figure 7b. Figure 7c and Figure 7d show the corresponding non-termination sensitive and timing sensitive control dependencies. Note, for example, that $7 \to_{\text{tscd}} 8$ because a choice $7 \to_{G} 11$ can delay node 8, but in contrast: $\neg(7 \to^{*}_{\text{ntscd}} 8)$, because no choice at node 7 can *prevent* node 8 from being executed. It is *not* the case that, in general, $n \to_{\text{ntscd}} m$ implies $n \to_{\text{tscd}} m$. For example: $2 \to_{\text{ntscd}} 8$, but $\neg(2 \to_{\text{tscd}} 8)$. What *does* hold here is $2 \to^{*}_{\text{tscd}} 8$ via $2 \to_{\text{tscd}} 7 \to_{\text{tscd}} 8$.

We will now provide a fixpoint characterization of $m \sqsubseteq^{k}_{\text{TIME}} n$. Remember from Theorem 2.1 that $\sqsubseteq_{\text{MAX}}$ is the least fixed point of the rule system D

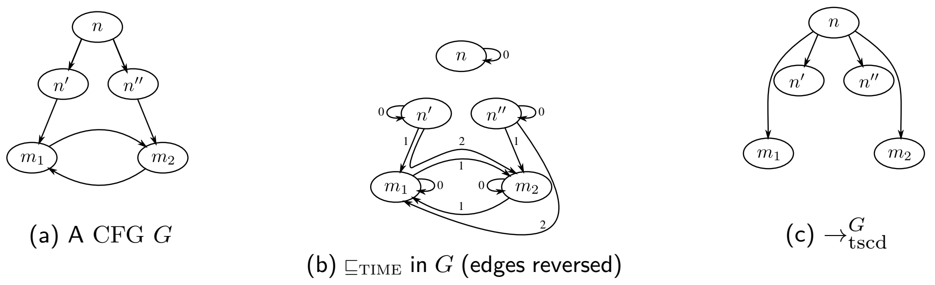
$$\frac{}{n \sqsubseteq n}\,\mathsf{D}^{\text{self}} \qquad \frac{\forall n \to x.\ m \sqsubseteq x \qquad n \to^{*} m}{m \sqsubseteq n}\,\mathsf{D}^{\text{succ}}$$

in the lattice $\left(2^{N \times N}, \subseteq\right)$. Similarly, the ternary relation $m \sqsubseteq^{k}_{\text{TIME}} n$ is the least fixed point of the rule system $\mathsf{T}_{\text{FIRST}}$ in the underlying lattice $\left(2^{N \times \mathbb{N} \times N}, \subseteq\right)$.

Theorem 3.1. Let $\mathsf{T}_{\text{FIRST}}$ be the rule-system

$$\frac{}{n \sqsubseteq^{0} n}\,\mathsf{T}^{\text{self}}_{\text{FIRST}} \qquad \frac{\forall n \to x.\ m \sqsubseteq^{k} x \qquad m \neq n \qquad n \to^{*} m}{m \sqsubseteq^{k+1} n}\,\mathsf{T}^{\text{succ}}_{\text{FIRST}}$$

Then $\sqsubseteq_{\text{TIME}} \ = \ \mu\mathsf{T}_{\text{FIRST}}$.



(a) A CFG G

(b) $\sqsubseteq_{\mathrm{TIME}}$ in G (edges reversed)

(c) $\to_{\mathrm{tscd}}^{G}$

Fig. 6. The canonical irreducible graph, where neither $n \to_{\mathrm{ntscd}} m_1$ nor $n \to_{\mathrm{ntscd}} m_2$.

Note that the condition $n \to^* m$ is redundant for nodes n that have *some* successor x, since we consider only the least, but not the greatest, fixed point of $\mathsf{T}_{\mathrm{FIRST}}$. The condition $m \neq n$ ensures that we only consider the first occurrence of m in each path.

We will now demonstrate that $\to_{\mathrm{tscd}}$ is *transitively* a stricter requirement than non-termination sensitive control independence. To this end, we use the following notation.

Definition 3.4. For $M \subseteq N$ and $\to$ a relation on N, the backward slice of M is

$$(\to)^* (M) = \{y \mid \exists x \in M : y \to^* x\}$$

This definition can be generalized to the ternary relation $\to$dod: if $y \to$dod (x_1, x_2), $y \in (\to\mathrm{dod})^* (M)$ only if x_1 *and* $x_2 \in (\to\mathrm{dod})^* (M)$ [29].

Theorem 3.2. Let $M \subseteq N$. Then

$$(\to_{\mathrm{tscd}})^* (M) \supseteq (\to\mathrm{ntscd} \uplus \to\mathrm{dod})^* (M)$$

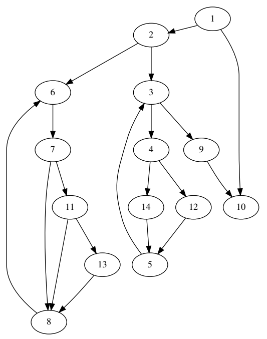
That is, there are more transitive time-sensitive CDs than the transitive closure of even the union of $\to_{\mathrm{ntscd}}$ and $\to$dod. Now remember that Ranganath and Amtoft introduced $\to$ntscd and $\to$dod in order to provide a sound notion of nontermination-sensitive backward slicing. Thus in the language of PDGs, $(\to)^* (M)$ is just the backward slice of M, and the theorem states that the timing sensitive backward slice of M contains the nontermination sensitive backward slice of M.

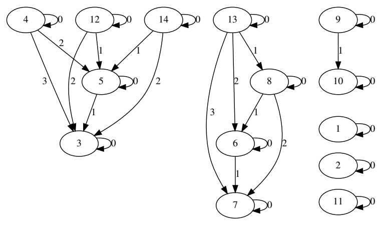
It is worth noting that the $\to_{\mathrm{tscd}}$ slice in Theorem 3.2 does *not* require a timing sensitive analogue of the relation $\to$dod. As seen above, the necessity of $\to$dod was motivated by an irreducible graph, such as the graph in Figure 6. But while in Figure 6 neither m_1 nor m_2 is nontermination sensitively control dependent on n, *both* m_1 and m_2 *are* timing-sensitively control dependent on n (e.g.: $n \to_{\mathrm{tscd}} m_1$ because $m_1 \sqsubseteq^1_{\mathrm{TIME}} n'$, but $\neg(m_1 \sqsubseteq^1_{\mathrm{TIME}} n'')$, and also: $m_1 \sqsubseteq^2_{\mathrm{TIME}} n''$, but $\neg(m_1 \sqsubseteq^2_{\mathrm{TIME}} n')$. This m_1/m_2 symmetry makes a ternary "$\to$tsdod" unnecessary.

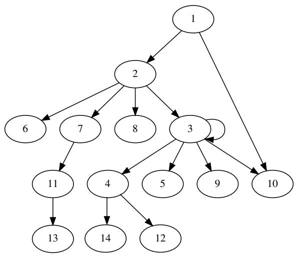
3.3 Soundness and minimality of $\to_{\mathrm{tscd}}$

It is our ultimate goal to discover timing leaks. We thus need a soundness proof for $\to_{\mathrm{tscd}}$, which guarantees that $\to_{\mathrm{tscd}}$ will indeed discover all potential timing leaks. We will further show that $\to_{\mathrm{tscd}}$ is minimal, which means there are no spurious time-sensitive dependencies.

Any soundness proof makes assumptions about the possibilities of attackers; this is called the attacker model. To prove soundness of $\to_{\mathrm{tscd}}$ under an attacker model, we use a technique called trace equivalence. Let us thus describe our attacker model, and then define trace


(a) CFG from Figure 2


(b) Its relation $\sqsubseteq_{\text{TIME}}$ (edges reversed)


(c) Its non-termination sensitive control dependence $\to_{\text{ntscd}}$

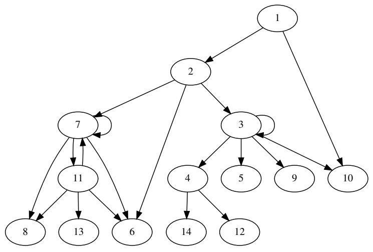

(d) Its timing sensitive control dependence $\to_{\text{tscd}}$

Fig. 7. Timing sensitive postdomination. Edges $n \xrightarrow{k} m$ indicate $m \sqsubseteq^{k}_{\text{TIME}} n$.

equivalence. We imagine an attacker who tries to infer secret values (such as `r2` in the AES example) measuring execution times for certain execution paths. But the attacker cannot observe all nodes, he can only observe certain "observable" nodes.[9] The goal of security analysis is then to guarantee that secret information cannot flow to observable nodes, resp. that execution times measured at observable nodes will not allow the attacker to infer secret values at unobservable nodes.[10] Technically, for $\to_{\text{tscd}}$ this guarantee is based on trace equivalence of *clocked traces.*

Definition 3.5. An (unclocked) trace t is a sequence of edges $(n, n') \in (\to_G) \cup (N_x \times \{\bot\})$ that is either finite with $t = (n_e, n_1),\ (n_1, n_2),\ \ldots,\ (n_k, n_x),\ (n_x, \bot)$ for some exit node $n_x \in N_x$, or infinite with $t = (n_e, n_1),\ (n_1, n_2),\ \ldots$. Partial edges $(n, \bot)$ occur only at exit nodes.

Definition 3.6. A clocked trace is a trace where every step is additionally annotated with a time stamp. We write t ⏱ $[i]$ if a trace step t has time stamp i. Given a trace $t = (n_e, n_1),\ (n_1, n_2),\ \ldots$, its clocked version is thus

$$t^{⏱} = (n_e, n_1) ⏱ [0],\ (n_1, n_2) ⏱ [1],\ \ldots$$

[9]These observable nodes are called "low" nodes in the literature on software security analysis (see e.g. [31]).
[10]This kind of security analysis is called *information flow control* (IFC), and is based on the technical notion of *noninterference*. We will describe technical details on the application of $\to_{\text{tscd}}$ for IFC in a separate article; here we present only the AES example and do not discuss technical details of noninterference.

Next, we assume there is a fixed set $S \subseteq N$ of observable nodes.[11]

Definition 3.7. Let $S \subseteq N$; let a trace t be given. We define the S-observation $t\big|_S$ of t to be the sub-sequence of t containing only edges (n, n') with $n \in S$. Traces t_1, t_2 are called S-equivalent if $t_1\big|_S = t_2\big|_S$.

These definitions work for unclocked and clocked traces. S-observability means that we assume an attacker to observe exactly those choices made at nodes $n \in S$. Specifically, we assume that an attacker can observe neither the nodes in a subtrace between observable nodes, nor – for unclocked traces – the *time spent* between two observable nodes (i.e: the *length* of the subtrace between two observable nodes).

Now we consider traces caused by specific inputs. We write t_i for the (possibly infinite) trace caused by input i. As we want to abstract away from particular input formats or data objects, we use a nonstandard formalization of input: i is a map from CFG nodes to a (perhaps infinite) list of CFG successor nodes: $i : N \to N^*$. An input i causes t_i as follows: If e.g. an `if` node $n \in N$ is visited for the kth time during the execution with input i, the execution will continue with the kth element of $i\,(n)$, which is a successor node (i.e. true or false path) of n. If n is only visited finitely often, superfluous entries in $i(n)$ are ignored.

This encoding has the effect that our CFGs are *state-free*: they contain CDs and nothing else. In particular the CFG does not contain program variables or program state – these are hidden in the i encoding. From a practical viewpoint this is however no restriction, and no weakening of the soundness property: we do not constrain possible i, and the soundness theorem below holds for all i, i'. Note however that for practical discovery of timing leaks, *data dependences* are additionally needed; this is described in section 3.4.

Next, we need the notion of S-equivalent inputs. For $S \subseteq N$, $i\big|_S : S \to N^*$ is the restriction of the map i to nodes $n \in S$, thereby only determining the successor nodes chosen at condition nodes $\in S$. Two inputs i, i' are called S-equivalent, written $i \sim_S i'$, if $i\big|_S = i'\big|_S$. An attacker cannot distinguish S-equivalent inputs.

We will now explain why – in the absence of timing leaks – S-equivalent inputs demand S-equivalent traces. It is essential to consider *clocked* traces: even if two unclocked traces are S-equivalent, their clocked versions may be different. This is the essence of time-sensitivity! For illustration consider Figure 5a, with observable nodes $S = \{m, m_x\}$. Regardless of the choice made at n, all inputs i, i' starting in m have the same observable trace

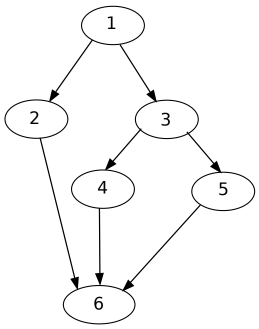
$$t_i\big|_S \;=\; (m, n)\,,\;(m_x, \perp) \;=\; t_{i'}\big|_S$$

Hence t_i and $t_{i'}$ are always trace equivalent. Thus an attacker without clock cannot extract any secret information from observing traces. However, if equipped with a suitably precise clock, an attacker will observe m_x after 5 steps for the input i that chooses n' at n, but already after 3 steps for i' that chooses n'' at n, exposing a timing difference. This becomes obvious if we use the clocked versions of t_i, $t_{i'}$, and then compare their S-observation:

$$\begin{array}{rcl} t^{⏱}_i\big|_S & = & (m, n)⏱\,[0]\,,\;(m_x, \perp)⏱\,[5] \\ & \neq & (m, n)⏱\,[0]\,,\;(m_x, \perp)⏱\,[3] \;=\; t^{⏱}_{i'}\big|_S \end{array}$$

Since the attacker cannot distinguish i and i' (they only differ in the choices for the unobservable node n), this timing difference allows the attacker to gain additional information, leading to a timing leak. On the other hand, the program in Figure 5b has no timing leak:

[11]The assumption of a fixed, static S and batch-like execution is standard in IFC and noninterference. It can be generalised and made more realistic in various ways; which however is not a topic of this article. Likewise, technical details of noninterference will not be discussed in this article.



i_1	$=$	$\{1 \to [3,3,\ldots], 3 \to [4,4,\ldots]\}$			
$t^{⏱}_{i_1}$	$=$	$(1,3)⏱[0], (3,4)⏱[1], (4,6)⏱[2], (6,\bot)⏱[3]$			
$t^{⏱}_{i_1}\big\vert_B$	$=$	$(1,3)⏱[0], (6,\bot)⏱[3]$	$t^{⏱}_{i_1}\big\vert_{B'}$	$=$	$(6,\bot)⏱[3]$

i_2	$=$	$\{1 \to [3,3,\ldots], 3 \to [5,5,\ldots]\}$			
$t^{⏱}_{i_2}$	$=$	$(1,3)⏱[0], (3,5)⏱[1], (5,6)⏱[2], (6,\bot)⏱[3]$			
$t^{⏱}_{i_2}\big\vert_B$	$=$	$(1,3)⏱[0], (6,\bot)⏱[3]$	$t^{⏱}_{i_2}\big\vert_{B'}$	$=$	$(6,\bot)⏱[3]$

i_3	$=$	$\{1 \to [2,2,\ldots], 3 \to [4,4,\ldots]\}$			
$t^{⏱}_{i_3}$	$=$	$(1,2)⏱[0], (2,6)⏱[1], (6,\bot)⏱[2]$			
$t^{⏱}_{i_3}\big\vert_B$	$=$	$(1,2)⏱[0]$	$t^{⏱}_{i_3}\big\vert_{B'}$	$=$	$(6,\bot)⏱[2]$

Fig. 8. In the CFG on the left, let $M = \{6\}$ be the slicing criterion. Then $B = BS\left(\{6\}\right) = \{1,6\}$ is the time-sensitive backward slice of M, because $1 \to_{\text{tscd}} 6$. $B' = \{6\}$ is a slice that is too small. Right: 3 different inputs with their traces and observable behaviour regarding B and B'.

even if we annotate each edge in the observable trace with its execution time, all inputs i, i' starting in m have the same observable *clocked* trace

$$t^{⏱}_i\big\vert_S \;=\; (m,n)⏱[0]\,,\;(m_x,\bot)⏱[5] \;=\; t^{⏱}_{i'}\big\vert_S$$

This discussion motivates the following definition of timing leaks:

Definition 3.8. Let $S \subseteq N$ be a set of observable ("low") nodes. A program is free of timing leaks if for all inputs i, i'

$$i \sim_S i' \implies t^{⏱}_i\big\vert_S = t^{⏱}_{i'}\big\vert_S$$

This definition is formally identical to classical noninterference definitions (cmp. e.g. [31]), but is based on clocked traces.

To prevent a timing leak, it is necessary that *all nodes which influence the timing of observable nodes* $\in S$ *are observable itself.* Otherwise, a secret node might influence the timing of an observable node. For example, Figure Figure 5a contains – as described above – a timing leak if we assume $S = \{m, m_x\}$. Indeed $n \to_{\text{tscd}} m_x$, but *not* $n \in S$. With $S' = S \cup \{n\} = \{m, n, m_x\}$, the timing leak disappears: While the timing of m_x still differs, i and i' are now distinguishable for the attacker, so this timing difference does not give additional information.

We will now show how $\to_{\text{tscd}}$ can be used to check for timing leaks. In particular we demonstrate that for any observable $M \subseteq S$, the time-sensitive backward slice $B = \left(\to_{\text{tscd}}\right)^* \left(M\right)$ fulfills the condition of definition 3.8. This implies that B is not too small, i.e. $\to_{\text{tscd}}$ is sound.

Before we state the theorem, consider what happens if B is too small. In that case, the $\to_{\text{tscd}}$ dependency would have "missing edges". Then there could exist two inputs that agree on B, but lead to different traces: $i \sim_B i'$ but $t_i\big\vert_B \neq t_{i'}\big\vert_B$. Figure 8 presents one such example. For $B = BS(\{6\}) = \{1,6\}$, we have $i_1 \sim_B i_2$ and indeed $t^{⏱}_{i_1}\big\vert_B = t^{⏱}_{i_2}\big\vert_B$. In contrast, the unsound "slice" $B' = \{6\}$ leads to $i_1 \sim_{B'} i_3$ but $t^{⏱}_{i_1}\big\vert_{B'} \neq t^{⏱}_{i_3}\big\vert_{B'}$. (Note that the only difference between the two slices is the timing of $\left(6,\bot\right)$, so we have $t_i\big\vert_{B'} = t_{i'}\big\vert_{B'}$ for the *unclocked* traces. In fact, $\{6\}$ is a sound slice when ignoring timing and using $\to_{\text{ntscd}}$.) If however $i \sim_B i'$ always implies $t^{⏱}_i\big\vert_B = t^{⏱}_{i'}\big\vert_B$, soundness is guaranteed.

Theorem 3.3 (Soundness of $\rightarrow_{\text{tscd}}$)**.** Let $M \subseteq S$. Let $B = \left(\rightarrow_{\text{tscd}}\right)^* \left(M\right)$ be the timing sensitive backward slice w.r.t M. Then, for any inputs i, i' such that $i \sim_B i'$, we have

$$t_i^{⏱}|_B = t_{i'}^{⏱}|_B$$

Corollary 3.1. If $BS(S) \subseteq S$, definition 3.8 holds, i.e. there is no timing leak.

As $S \subseteq BS(S)$ always holds, the corollary's premise is in fact $S = BS(S)$. If the premise is not satisfied, i.e. for some $x \in BS(S)$: $x \notin S$, x – as explained above – is a timing leak.

Minimality of slicing now shows that $B = BS\left(M\right)$ is as small as possible: Any set of nodes B' that includes the slicing criterion M can only be secure if it is a superset of B.

Theorem 3.4 (Minimality of $\rightarrow_{\text{tscd}}$)**.** Under the assumptions of Theorem 3.3, for any $B' \supseteq M$ with $B' \not\supseteq B$ there exist inputs i, i' such that $i \sim_{B'} i'$, but:

$$t_i^{⏱}|_{B'} \neq t_{i'}^{⏱}|_{B'}$$

It should be noted that the proof for both theorems relies on the non-standard, state-free input encoding of i, i', which was described above.

3.4 The Full Time-Sensitive Backward Slice

Our nonstandard input encoding (which "factors away" all state information) is not practical for "real" programs. In such programs, time-sensitive influences through variables must be considered too. For this reason, discovery of timing leaks needs data dependences in addition to control dependences. Data dependences have in fact already been used in the AES example. For completeness and better understanding, we will thus describe the full algorithm for discovering timing leaks. Note that in this article, we do not provide a modified soundness proof for the full algorithm, as it does not contribute to $\rightarrow_{\text{tscd}}$ "as such".

We denote data dependencies by $\rightarrow_{dd}$. $x \rightarrow_{dd} y$ means that a variable v which is defined (assigned) at x is used at y; provided there is a CFG path $x \rightarrow^* y$, and v is not redefined on this path [11]. We will not describe the construction of $\rightarrow_{dd}$ in detail, but note that for full languages with functions, objects, multithreading etc. the computation of precise data dependencies is nontrivial and requires context-sensitive summary dependencies, precise points-to analysis, may-happen-in-parallel analysis, and much more (see e.g. [14, 23, 30]).

The full algorithm for discovering timing leaks then assumes $\rightarrow_{dd}$, and proceeds as follows.

(1) Compute $\rightarrow_{\text{tscd}}$. If $x \rightarrow_{\text{tscd}} y$, but *not* $x \rightarrow_{cd} y$, then there may be a timing leak at y, but only if it can be influenced by secret data.

(2) Using $\rightarrow_{dd}$, the full time-sensitive backward slice is defined as

$$BS_{ts}(M) = \left(\rightarrow_{\text{tscd}} \cup \rightarrow_{dd}\right)^* \left(M\right)$$

This slice contains all CFG nodes which may influence M; other nodes which influence M cannot exist [6, 14, 18].

(3) Now if $x \rightarrow_{\text{tscd}} y$, and $BS_{ts}(\{x\})$ contains any secret input or variables, there is a timing leak at y: the execution time between x and y varies, depending on secret data.

This procedure is fully analogous to the slicing-based noninterference check used in JOANA (see [6, 14]; these papers include soundness proofs and other details about slicing-based IFC), but with $\rightarrow_{\text{tscd}}$ instead of $\rightarrow_{cd}$. Note that in the current article, we consider only context-insensitive timing-dependencies (while JOANA uses context-sensitive, object-sensitive dependencies). A context-sensitive $\rightarrow_{\text{tscd}}$ is future work.

4 TIMING SENSITIVITY FOR MICROARCHITECTURAL CFGS

Already in the abstract we mentioned the infamous AES cache timing leaks which where discovered by Bernstein [4]. Some details of this attack were described in section 3.1. We will now describe in more detail how such cache leaks can be discovered resp. prevented via time-sensitive CDs in microarchitectural CFGs. Basically, the algorithm from section 3.4 is used, but the underlying CFG must be extended to model cache behaviour.

In the following, we describe this cache-modelling CFG extension in detail. The CFG edges are labeled with assignments and guards that refer to (cacheable) variables $\mathtt{a}, \mathtt{b}, \ldots$, and uncacheable registers $\mathtt{r1}, \mathtt{r2}, \ldots$.

We assume a simple data cache of size four, with a least recently used eviction strategy. The (micro-architectural) cache-state hence consists of a list $\left[x_1, x_2, x_3, x_4\right]$ of variables, with x_1 being the most recently used, and x_4 the next to be evicted. In Figure 9b, we show — under an abstraction that considers cache state only — all possible executions of the control flow graph, assuming an empty initial cache. For example, the abstract node $\left(9, \left[\mathtt{x}, \mathtt{d}, \mathtt{c}, \mathtt{b}\right]\right)$ represents all those concrete configurations at control node 9 in which the concrete micro-architectural cache contains cached values for the variables $\left[\mathtt{x}, \mathtt{d}, \mathtt{c}, \mathtt{b}\right]$, in this order (with arbitrary concrete macro-architectural state).

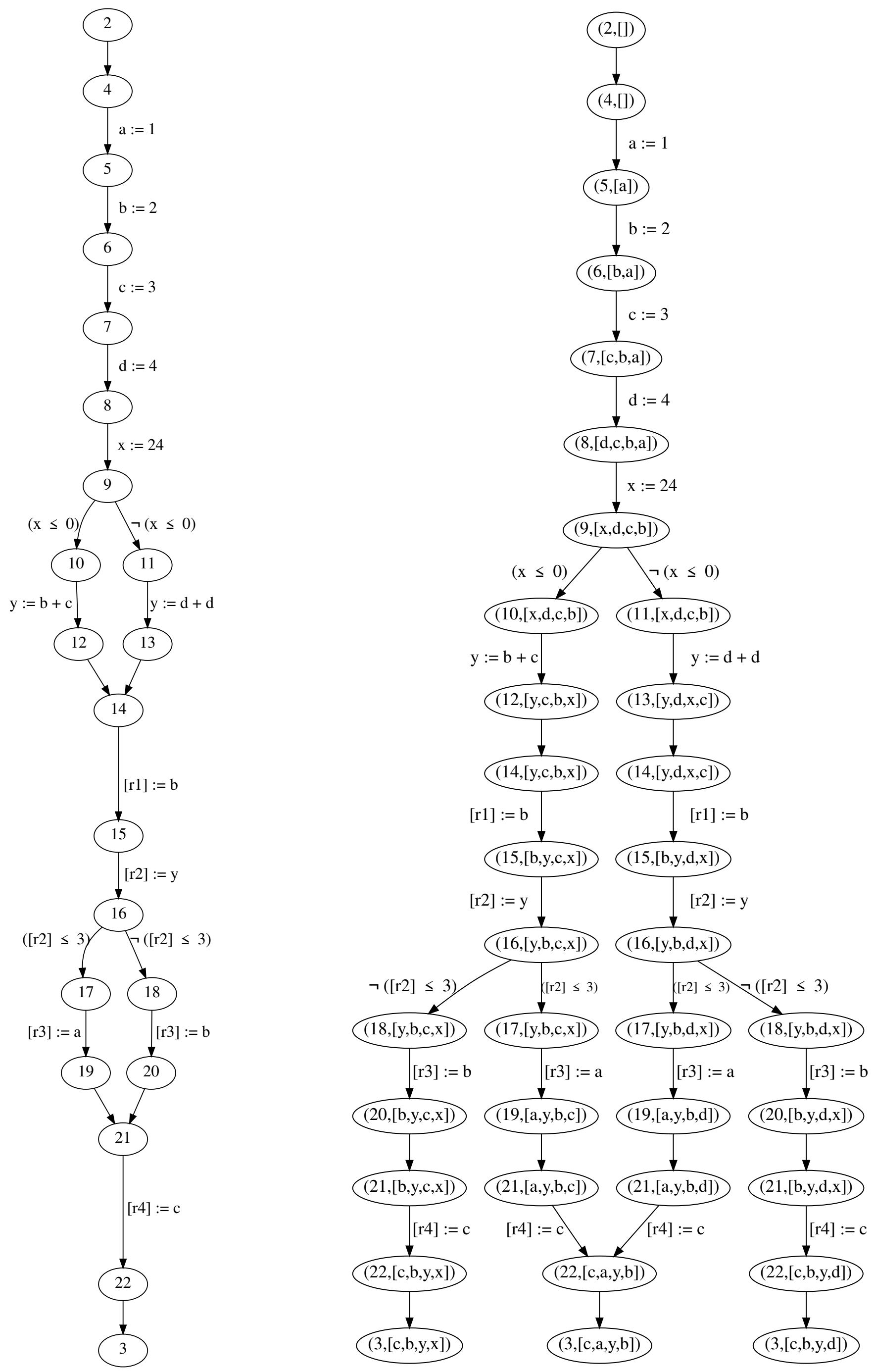
In the example, executions can reach the control node $m = 15$ at cache states represented by either $\left[\mathtt{b}, \mathtt{y}, \mathtt{c}, \mathtt{x}\right]$, or by $\left[\mathtt{b}, \mathtt{y}, \mathtt{d}, \mathtt{x}\right]$. Which of these (abstract) cache states is reached is determined by the macro-architectural choice made at $n = 9$. But it is easy to see that the execution time of the read of $\mathtt{y}$ at node $m = 15$ does *not* depend on the choice made at $n = 9$, since in both (classes of) executions that reach node $m = 15$, the cache *does* contain the variable $\mathtt{y}$, which is the only cacheable variable accessed by the edge $15 \xrightarrow{\mathtt{r_2:=y}} 16$ at m.

For the read of variable $\mathtt{b}$ at node $m = 14$, on the other hand, one class of executions reaches m in $\left(14, \left[\mathtt{y}, \mathtt{c}, \mathtt{b}, \mathtt{x}\right]\right)$ (containing $\mathtt{b}$), while another class of executions reaches m in $\left(14, \left[\mathtt{y}, \mathtt{d}, \mathtt{x}, \mathtt{c}\right]\right)$ (*not* containing $\mathtt{b}$). Whether the relevant variable $\mathtt{b}$ is in the cache at $m = 14$ (and hence: the execution time of the read of $\mathtt{b}$ at $m = 14$) or not depends here on the choice made at $n = 9$.

Now consider the read of $\mathtt{c}$ at node $m = 21$. Does its cache state depend on the choice made right before at $n' = 16$? There are four (abstract) cache states at $m = 21$. Two contain the variable $\mathtt{c}$: $\left(21, \left[\mathtt{b}, \mathtt{y}, \mathtt{c}, \mathtt{x}\right]\right)$ and $\left(21, \left[\mathtt{a}, \mathtt{y}, \mathtt{b}, \mathtt{c}\right]\right)$. The other two do *not* contain $\mathtt{c}$: $\left(21, \left[\mathtt{a}, \mathtt{y}, \mathtt{b}, \mathtt{d}\right]\right)$ and $\left(21, \left[\mathtt{b}, \mathtt{y}, \mathtt{d}, \mathtt{x}\right]\right)$. The cache states containing $\mathtt{c}$ are reachable from configurations at control node $n' = 16$. At the same time: cache states *not* containing $\mathtt{c}$ are *also* reachable from configurations at control node $n' = 16$. But in fact, whether $\mathtt{c}$ is in cache at m does *not* depend on the choice made at n'. To see this, note that node $n' = 16$ can be reached at two different cache states. The first abstract configuration is $\left(16, \left[\mathtt{y}, \mathtt{b}, \mathtt{c}, \mathtt{x}\right]\right)$. But whenever $m = 21$ is reached from this abstract configuration, $\mathtt{c}$ *is* in the cache (either $\left(21, \left[\mathtt{b}, \mathtt{y}, \mathtt{c}, \mathtt{x}\right]\right)$ or $\left(21, \left[\mathtt{a}, \mathtt{y}, \mathtt{b}, \mathtt{c}\right]\right)$). The second abstract configuration at which $n' = 16$ can be reached is $\left(16, \left[\mathtt{y}, \mathtt{b}, \mathtt{d}, \mathtt{x}\right]\right)$. But whenever $m = 21$ is reached from that configuration, $\mathtt{c}$ *is not* in the cache ($\left(21, \left[\mathtt{a}, \mathtt{y}, \mathtt{b}, \mathtt{d}\right]\right)$ or $\left(21, \left[\mathtt{b}, \mathtt{y}, \mathtt{d}, \mathtt{x}\right]\right)$).

On the other hand, the cache status of $\mathtt{c}$ at node $m = 21$ *does* depend on the choice made earlier at $n = 9$. In this example this is necessarily so, since the node $n = 9$ is the only other macro-architectural conditional node in the control flow graph. But this is also directly evident by the structure of the graph in Figure 9b.

Note that through a a small modification of the program, the cache status of $\mathtt{c}$ at $m = 21$ could have been *independent* from the choice made earlier at $n = 9$. For example: had there been reads to two additional variables (e.g: $\mathtt{e}, \mathtt{f}$) right before $m = 21$, then *all* cache states



(a) Control Flow Graph (b) Cache Aware Abstract Executions

Fig. 9. A CFG and its possible cache-aware abstract executions.

at m would *not* have contained `c`. This is because these two reads would have evicted `c` even from $\left[\mathtt{b}, \mathtt{y}, \mathtt{c}, \mathtt{x}\right]$ (and $\left[\mathtt{a}, \mathtt{y}, \mathtt{b}, \mathtt{c}\right]$).

In summary, the choice made at $n = 9$ does influence the relevant (micro-architectural) cache state at $m \in \{21, 14\}$. In fact for this micro-architecture, these are the *only* micro-architectural dependencies in this CFG. The example indicates how a CFG G can be transformed into a cache-aware version. We will not present the formal definitions here (see [16]), but just present the transformed CFG for the above example.

Figure 10 shows the micro-architectural-aware CFG G' for Figure 9; together with an explicit timing cost model C'. A cache-miss is assumed to take 10 units of time, while a cache-hit takes 2 units[12]. At node 14, the read from `b` takes either 2 or 10 units of time, since `b` there might either be in the cache, or not.[13] Hence in G', node 14 has *two* artificial successors: the read from `b` takes either 2 or 10 units of time, since `b` there might either be in the cache, or not. On the other hand, node 15 still has only one successor, reached with timing cost $3 = 2 + 1$ (cache access plus register access), since we found that there the variable `y` is always in cache.

In G', we now have (as desired) that node 21 is in the backward slice of the exit node 3. Formally,

$$21 \in \left(\rightarrow_{\text{tscd}}^{G'\left[C'\right]}\right)^{*}(\{3\})$$

Together with the microarchitectural dependence from node 9 to node 21, we conclude that the decision at node 9 may influence the execution time of node 3.

Note that even if G is deterministic, G' usually is not. This is no problem, because we can still use the micro-architectural dependencies $\rightarrow_{\mu\text{d}}^{G}$ (and data dependencies $\rightarrow_{dd}$) from the original graph G, and only use G' for timing sensitive control dependence $\rightarrow_{\text{tscd}}^{G'\left[C'\right]}$.

For the AES code, the cache-sensitive graph G' has been shown in section 3.1, and we already described how cache leaks in AES have been discovered through time-sensitive backward slicing. More details can be found in [16].

5 ALGORITHMS

Our algorithms are based on the fundamental insight that Cytron's original algorithm for dominance frontiers can be generalized to CFGs with loops and multiple exit nodes; and even to the computation of time-sensitive CD. We consider this "generic" algorithm our major contribution: without it, the new $\rightarrow_{\text{tscd}}$ definition would be worthless in practice; and even Ranganath's and Amtoft's $\rightarrow_{\text{ntscd}}/\rightarrow_{\text{nticd}}$ are more efficient to compute using the new algorithms.

5.1 New algorithms for $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$

Let us begin with new algorithms for $\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{nticd}}$. These will – in generalization of Cytron's approach – be constructed as postdominance frontiers of $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$. The efficient implementation of $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ needs some technical machinery, namely transitive reductions and pseudo-forests.

Both $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ will always be represented by their transitive reductions; allowing efficient construction algorithms. A transitive reduction $<$ of a transitive relation $\sqsubseteq$ is

[12]memory writes are assumed to always take 2 units of times, and register accesses take 1 unit of time
[13]In the timing cost model C, the cost $11 = 10 + 1$ that stems from one uncached variable access plus one register access is split into two edges. We need to do this because in our notion of graphs, there can be no multi-edges, and we require cost models C to be *strictly* positive.

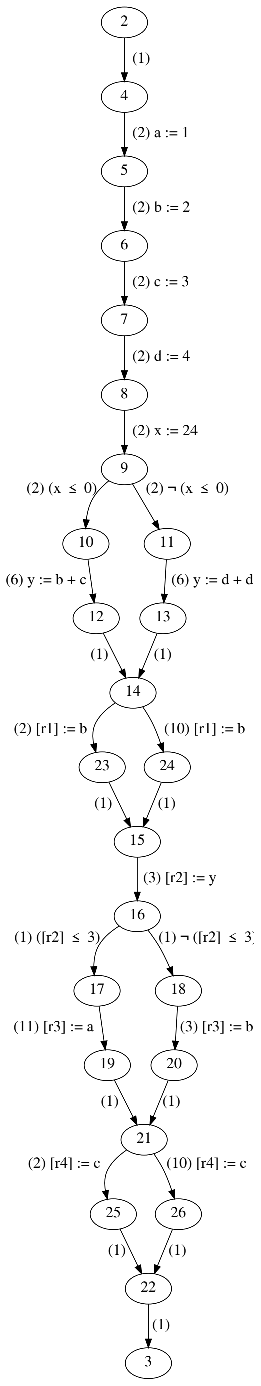
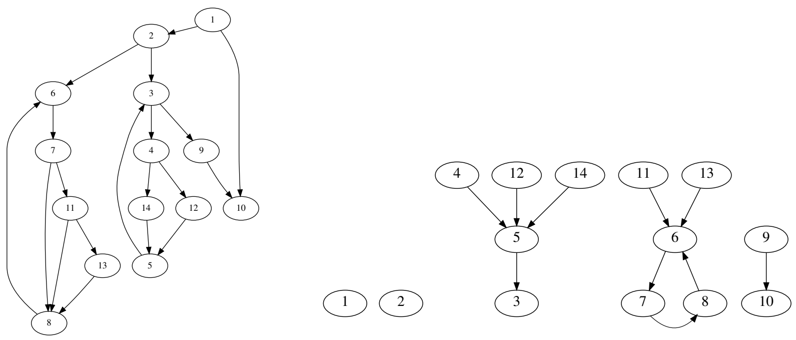


Fig. 10. Micro-Architecture Aware CFG G' for Figure 9.



(a) CFG G from Figure 2 (b) Transitive Reduction $<_{\text{MAX}}$ of $\sqsubseteq_{\text{MAX}}$

Fig. 11. Nontermination-sensitive Postdominance

a minimal subset $<$ of $\sqsubseteq$ such that $(<)^* = \sqsubseteq$. Thus $<$ has a minimal number of edges but the same transitive closure as $\sqsubseteq$. Efficient algorithms for transitive reductions have long been known [2]. But remember that $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ may contain cycles (i.e. are not antisymmetric), in contrast to the classical $\sqsubseteq_{\text{POST}}$. Hence their transitive reductions may also contain cycles. Therefore the transitive reductions of $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ are not forests (i.e. sets of trees) as for $\sqsubseteq_{\text{POST}}$, but so-called *pseudo-forests.*

Definition 5.1. A pseudo-forest is a relation $<$ such that for every node $n \in N$, $m < n$ for *at most one* node m.

Thus in a pseudo-forest every node has at most one parent node, but in contrast to ordinary forests, pseudo-forests may contain cycles. Summarizing this discussion, we obtain

Observation 5.1. 1. Both $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ are reflexive and transitive, but not necessarily anti-symmetric.
2. Any transitive, reflexive reduction $<_{\text{MAX}}$ of $\sqsubseteq_{\text{MAX}}$ is a pseudo-forest.
3. Any transitive, reflexive reduction $<_{\text{SINK}}$ of $\sqsubseteq_{\text{SINK}}$ is a pseudo-forest.

Figure 11 (b) shows a reduction $<_{\text{MAX}}$ of $\sqsubseteq_{\text{MAX}}$ for the CFG in Figure 11 (a). This pseudo-forest has five trees, with roots 1, 2, 3, $\{6, 7, 8\}$ and 10.[14] Node 9 does *not* $\sqsubseteq_{\text{MAX}}$-postdominate node 3 because the loop at 3 may not terminate. On the other hand, node 9 *does* $\sqsubseteq_{\text{SINK}}$-postdominate node 3: a path looping forever at 3 is *not* a sink path, and any sink path starting at 3 must eventually reach the trivial sink at node 10.

We will now present new algorithms to compute $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$. The representation of both $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ by pseudo-forests is crucial, as pseudo-forests admit efficient algorithms for their computation. Based on pseudo-forests, our algorithm for $\sqsubseteq_{\text{MAX}}$ is a standard fixpoint iteration. Beginning with the empty pseudo-forest, new edges are added to $<_{\text{MAX}}$ according to Theorem 2.1 until a fixpoint is reached. Since $\sqsubseteq_{\text{MAX}}$ is efficiently represented by a pseudo-forest $<_{\text{MAX}}$, it is straightforward to derive an efficient algorithm for the computation of $\sqsubseteq_{\text{MAX}}$, see algorithm 2. In addition, we need an efficient implementation of set-intersection

[14] In the figure, downarrows $n \to m$ mean that $m < n$.

```
Input   : A pseudo-forest $<$, represented as a map IMDOM : $N \hookrightarrow N$ s.t.
          IMDOM $[n] = m$ iff $m < n$.
Input   : Nodes $m_0$, $n_0$
Output  : A least common ancestor of $n_0, m_0$, or $\perp$ if there is none.
begin
    return lca $(n_0,\ m_0)$
end
Function lca $(\pi_n,\ \pi_m)$
    Input   : A $<$-path $\pi_n = n_0, \ldots, n$ ending in $n$
    Input   : A $<$-path $\pi_m = m_0, \ldots, m$ ending in $m$
    if $m \in \pi_n$ then return $m$
    switch IMDOM[n] do
        case $\perp$ do return lin$[\pi_n]$ $(\pi_m)$
        case n' do
            if $n' \in \pi_n$ then
                return lin$[\pi_n]$ $(\pi_m)$
            end
            return lca $(\pi_m,\ \pi_n n')$
        end
    end
end
Function lin$[\pi_n]$ $(\pi_m)$
    Input    : A $<$-path $\pi_m = m_0, \ldots, m$ ending in $m$
    Implicit : A $<$-path $\pi_n = n_0, \ldots, n$ ending in $n$
    switch IMDOM[m] do
        case $\perp$ do return $\perp$
        case m' do
            if $m' \in \pi_n$ then return m'
            if $m' \in \pi_m$ then return $\perp$
            return lin$[\pi_n]$$(\pi_m m')$
        end
    end
end
```

Algorithm 1: A *least common ancestor* algorithm for pseudo-forests. $N \hookrightarrow N$ denotes a partial map from N to N.

in the representation $<$, i.e.: a *least common ancestor* algorithm lca$_<$ for pseudo-forests; see algorithm 1.

Algorithm 1 calculates the least common ancestor of n_0 and m_0 by alternately extending $<$-paths from n_0 and m_0 one by one. If the newly added node is already contained in the other path, it is returned as the result of `lca`(n_0, m_0): Since this is the first time the two paths overlap, this node is not only a common ancestor but also the least one. If one path cannot be extended (because its `IMDOM` is $\perp$ or starts to contain a cycle), only the other path is extended (procedure `lin`). When the other path cannot be extended anymore either, we do not have an `lca`, so we return $\perp$.

Algorithm 2 works in two phases: First, we establish trivial `IMDOM` relations for nodes with only one successor. For the graph in Figure 14 (left), these would be `IMDOM[5]=7`, `IMDOM[7]=8`, `IMDOM[8]=9` and `IMDOM[9]=8`.

Next, we calculate `IMDOM` for conditional nodes (nodes with more than one successor). We keep a queue of such nodes for which `IMDOM` has not been calculated. For each conditional node `x`, we try to calculate the `lca` of their successors. If this returns a node `a` $\neq \perp$, we set `IMDOM[x]=a` and remove `x` from the worklist, otherwise it is put back at the end. The algorithm terminates once the worklist is empty or we have completed a full iteration through the worklist without a change to `IMDOM`. The variable `oldest` tracks the first node after the last change; once we visit it again, we are done.

In the case of the graph in Figure 14 (left), assume we iterate in order COND_G `=[1,2,3,4]`, which becomes our first `workqueue`. For `x=1`, we calculate `lca({2, 3, 4})`. Let's suppose we try to calculate `lca(2, 3) = lca([2], [3])` first. Since `IMDOM[2]=`$\perp$, we immediately call `lin[[2]]([3])`, but since `IMDOM[3]=`$\perp$ as well, we return $\perp$ as `lca(2, 3)`. But then `lca({2, 3, 4})=`$\perp$ as well. `1` is therefore put back into the queue, so we now have `workqueue=[2,3,4,1]` and `oldest=1`. For `x=2`, when calculating `lca(6, 7)`, we have `IMDOM[6]=`$\perp$, so we immediately call `lin[[6]]([7])`. During the recursion in `lin`, we extend `[7]` to `[7,8]` and `[7,8,9]` (since no new node is in `[6]`). The next step would be `IMDOM[9]=8`. Since `8` $\in$ `[7,8,9]` (which would lead to a loop), we return $\perp$. `2` is therefore put back into the queue, so we now have `workqueue=[3,4,1,2]` and `oldest=1`. For `x=3`, when calculating `lca(5, 7)`, we have `IMDOM[5]=7` and `7` $\in$ `[7]`, so we return `7` as our `lca`. Since we have an `lca`, we update `IMDOM[3]=7`, keep `3` out of the workqueue (so `workqueue=[4,1,2]`) and set `oldest=`$\perp$. For `x=4`, when calculating `lca(9, 5)`, we extend both paths alternately until the path starting in `9` would enter a loop, then only the path starting in `5` is extended. Then we will find that `8` is our `lca`, since it is in both paths. In detail, `lca([9], [5])` $=$ `lca([5], [9,8])` $=$ `lca([9,8], [5,7])` $=$ `lin[[9,8]]([5,7])` $=$ `lin[[9,8]]([5,7,8])` $=$ `8`. We update `IMDOM[4]=8`, keep `4` out of the workqueue (so `workqueue=[1,2]`) and keep `oldest=`$\perp$. Now we are back to `x=1`. When calculating `lca(2, 3)`, we now have `IMDOM[3]=7`, so we can extend `[3]` until we get `[3,7,8,9]`. But still, no node is contained in `[2]`, so we still have $\perp$ as our `lca`. We put `1` back into the workqueue (so `workqueue=[2,1]`) and set `oldest=1`. After finding for `x=2` that `lca(6,7)=` $\perp$, we put `2` back into the workqueue (so `workqueue=[1,2]`) and keep `oldest=1`. But now our next element in the queue is our `oldest`, so we are done.

The computation of $\sqsubseteq_{\text{SINK}}$ is slightly more complicated. As it is a greatest fixpoint, in principle we must start with $N \times N$ and reduce it according to the rules; until the greatest fixpoint is reached. But $N \times N$ cannot be represented by a pseudo-forest. Hence we need to initialize the fixed point iteration with an approximation $\sqsubseteq_0$ of $\sqsubseteq_{\text{SINK}}$ (i.e.: $\sqsubseteq_0 \;\supseteq\; \sqsubseteq_{\text{SINK}}$) that is representable by a pseudo-forest $<_0$. We can build $<_0$ by interleaving a traversal of a preliminary pseudo forest $<$ with $\text{lca}_<$ computations. Consider the preliminary $<$ in figure 12b. We need to establish $3 < 1$, but find that $\text{lca}_< \big(\{2, 3\}\big) = \perp$ for the successors of 1. We would like to assume *both* $3 < 1$ and $2 < 1$, the latter of which would then be invalidated in the (downward) fixed point iteration. But then $<$ no longer would be a pseudo forest! If we assumed just $2 < 1$, we would obtain a $<_0$ such that *not*: $<_0^* \;\supseteq\; \sqsubseteq_{\text{SINK}}$, so we need to make the assumption $3 < 1$. This example illustrates how the fixpoint iteration must proceed. It is based on the

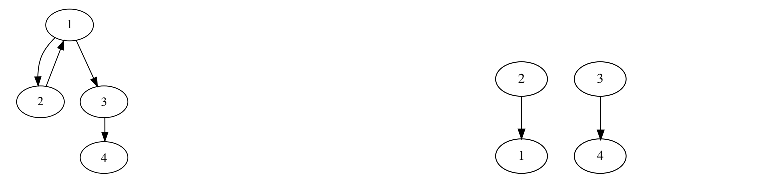
Observation 5.2. Let $<_{\text{SINK}}$ be a transitive reduction of $\sqsubseteq_{\text{SINK}}$. Then whenever $x <_{\text{SINK}} y$ and any path starting in x is bound for a sink S (such S is necessarily unique), then any path starting in y is bound for S as well.

```
Input   : A CFG $G$
Data: A pseudo-forest $<$ represented as a map IMDOM $: N \hookrightarrow N$ s.t. IMDOM $\left[n\right] = m$ iff
      $m < n$
Output: A transitive reduction $<_{\text{MAX}}$ of $\sqsubseteq_{\text{MAX}}$
begin
    for $x \in N,\ \{z \mid x \to z\} = \{z\},\ z \neq x$ do
        IMDOM $\left[x\right] \leftarrow z$
    end
    MAXIMAL$_{\text{up}}$
    return IMDOM
end
Procedure MAXIMAL$_{\text{up}}$
    workqueue $\leftarrow$ COND$_G$
    oldest $\leftarrow \bot$
    while workqueue $\neq \emptyset$ do
        $x \leftarrow$ removeFront(workqueue)
        assert IMDOM$[x] = \bot$
        if oldest $= x$ then
            return
        end
        if oldest $= \bot$ then
            oldest $\leftarrow x$
        end
        $a \leftarrow$ lca $\left(\{\, y \mid x \to y \,\}\right)$
        $z \leftarrow \begin{cases} \bot & \text{if } a = \bot \ \vee \ a = x \\ a & \text{otherwise} \end{cases}$
        if $z \neq \bot$ then
            IMDOM $\left[x\right] \leftarrow z$
            oldest $\leftarrow \bot$
        else
            pushBack $\left(\text{workqueue}, x\right)$
        end
    end
end
```

Algorithm 2: An efficient algorithm for the computation of $<_{\text{MAX}}$. COND$_G$ denotes the set of *conditional* nodes, i.e.: nodes with more than one successor. workqueue is ordered by any fixed ordering on nodes N.

Here, "bound for S" means that the path cannot escape sink S. To illustrate the iteration for $\sqsubseteq_{\text{SINK}}$, consider figure 12b. For node 3 we have already established $4 <^* 3$ for the sink node $4 \in S$, but we have not yet established $4 <^* 2$. This suggests that we must – whenever $\text{lca}_< \left(\{\, y \mid x \to y \,\}\right) = \bot$ – choose some successor node y of x such that already $s <^* y$ for some sink node s. We call such nodes y *processed*, and maintain a set PROCD of all such nodes. Algorithm 3 presents the computation of $<_{\text{SINK}}$, the additional procedures performing the iteration are given in Figure 13.

Algorithm 3 first initializes ISDOM for sink nodes and nodes with one successor. Remember that any nontrivial sink S_i contains a $<_{\text{SINK}}$ -cycle. For each sink S_i, we therefore initialize ISDOM to be such a cycle in arbitrary order. We also choose a representative s_i for each sink



(a) A graph G (b) A preliminary pseudo forest $<$

Fig. 12. Computing an initial approximation $<_0$.

S_i and mark all nodes in S_i as processed. For all nodes outside sinks with one successor, the initialization of `ISDOM` is identical to the one in algorithm 2. Once a successor is processed, we mark all nodes that reach this node through `ISDOM`-chains as processed.

Next, we construct in $SINK_{up}$ a preliminary `ISDOM` that fulfills `ISDOM`$^* \supseteq \sqsubseteq_{\text{SINK}}$ but might be too optimistic: For nodes `x`, `ISDOM[x]` might exist even though it should be $\perp$; or it might be a node that is too small to be a common ancestor of the successors of `x` (but the correct `lca` is an ancestor of `ISDOM[x]`). We choose such an `ISDOM` of `x` as soon as one of its successors is processed. When calculating the `lca`, we only consider the successors which have already been processed. If the resulting `lca` is $\perp$, we choose an arbitrary successor as `lca`. Now, we set `ISDOM[x]` to be this `lca`. We also set `x` and all nodes that reach `x` through `ISDOM`-chains as processed. This succeeds for any `x` with distance k to a sink at attempt k at the latest (this can be shown by induction on k), so this algorithm terminates.

Then, these spurious postdominances are eliminated during $SINK_{down}$. For each conditional node `x` outside sinks the `lca` of its successors is calculated. If it is part of a sink S_i, its distinguished representative s_i is chosen instead. If it is different from the current `ISDOM` (either a different node or $\perp$), `ISDOM` is updated and all nodes possibly affected by this change are put back in the worklist: These are all conditional nodes `n` having a successor `y` that reaches `x` through `ISDOM`-chains. This is done until the worklist is empty.

As an example, consider Figure 14 (left). In the initial phase, we set `ISDOM[8]=9` and `ISDOM[9]=8` for the non-trivial sink. For its representative, let's assume we choose `8`. We mark all sink nodes `6`, `8` and `9` as processed. Then, we handle non-condition nodes. We set `ISDOM[4]=9` and mark `4` as processed. After that, we set `ISDOM[5]=7` (but cannot mark it as processed since `7` is not). Finally we set `ISDOM[7]=8` and mark both `7` and `5` as processed.

In $SINK_{up}$, `1` has a single processed successor, namely `4`. Thus `ISDOM[1]=4`, and `1` is processed. For `2`, we have two processed successors, but `lca(6,7)=`$\perp$. Let's suppose we choose `ISDOM[2]=7`; `2` is also marked as processed. Finally, `3` has two processed successors and `lca(5,7)=7`, so we set `ISDOM[3]=7` and mark `3` as processed. This finishes $SINK_{up}$.

In $SINK_{down}$, we first check `x=1`. Since we still have `ISDOM[2]=7`, `lca({2,3,4})=8`. This is also the representative of this sink, so we set `ISDOM[1]=8`. For `x=2`, we now find that `ISDOM[2]=`$\perp$. This change puts `1` back into the worklist. For `x=3`, no change occurs, since `lca(5,7)=ISDOM[3]=7`. For `x=4`, we have `lca(9)=9`. The representative of this sink is `8`, so we set `ISDOM[4]=8`. We would also have to put `1` back into the worklist if it wasn't there already. For `x=1`, the updated `ISDOM[2]` now means we find `lca({2,3,4})=`$\perp$, so we set `ISDOM[2]=`$\perp$. This finishes the calculation of `ISDOM`.

5.2 Postdominance Frontiers in Graphs Without Unique Exit

We will now derive algorithms for $\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{nticd}}$, based on $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$. In particular, we generalize Cytron's idea to split up the postdominance frontier into an "up" and a "local"

```
Input  : A CFG $G$
Data: A pseudo-forest $<$ represented as a map $\mathsf{ISDOM} : N \hookrightarrow N$ s.t. $\mathsf{ISDOM}\left[n\right] = m$ iff
      $m < n$
Output: A transitive reduction $<_{\text{SINK}}$ of $\sqsubseteq_{\text{SINK}}$
begin
    $\{S_1, \ldots, S_n\} \leftarrow \left\{S_i \;\middle|\; S_i \in \mathsf{sccs}\left(G\right), \neg\, \exists s \to n.\ s \in S_i \ \wedge\ n \notin S_i\right\}$
    $S \leftarrow \bigcup S_i$
    for $1 \leq i \leq n$ do
        $s_i \leftarrow$ any node in $S_i$
        for $n_j \in S_i$ in any fixed ordering $n_1, \ldots, n_{k_i}$ of $S_i$ do
            $\mathsf{ISDOM}\left[n_j\right] \leftarrow n_{j+1 \mod k_i}$ unless $k_i = 1$
            $\mathsf{processed}\left(n_j\right)$
        end
    end
    for $x \in N,\ x \notin S,\ \{z \mid x \to z\} = \{z\},\ z \neq x$ do
        $\mathsf{ISDOM}\left[x\right] \leftarrow z$
        if $z \in \mathsf{PROCD}$ then $\mathsf{processed}\left(x\right)$
    end
    $\text{SINK}_{\text{up}}$
    $\text{SINK}_{\text{down}}$
    return $\mathsf{ISDOM}$
end
```

Algorithm 3: Computation of transitive reduction $<_{\text{SINK}}$ of $\sqsubseteq_{\text{SINK}}$. Not shown is the procedure processed $\left(x\right)$ which updates PROCD given a node x s.t. $s <^* x$ for some sink node s, by following linear segments ending in x upwards.

part, and to follow the tree structure (parent links) while iterating. The latter also works for pseudo-forests.

To describe this idea in detail, first remember that in graphs with unique exit node n_x, standard postdominance $\sqsubseteq_{\text{POST}}$ is always a partial order, while in arbitrary graphs, $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ may lack anti-symmetry, and may thus contain cycles of nodes postdominating each other. In the following we therefore reconstruct Cytron's algorithm with our generalized definition for postdominance frontiers. In particular, the following definitions replace Cytron's definitions from [10]: instead of Cytron's original $\sqsubset_{\text{POST}}$ we use our new $1\text{-}\sqsubseteq$, and instead of Cytron's original $\text{ipdom}_{\sqsubseteq_{\text{POST}}}$ we use $\text{ipdom}_{\sqsubseteq}$. We will thus be able to define the generalized algorithm in a self-contained way.

Definition 5.2 (Immediate $\sqsubseteq$-Postdominance)**.** Given a binary relation $\sqsubseteq$ on nodes, a node x is said to $1\text{-}\sqsubseteq$-postdominate z if there exists some node $y \neq x$ such that $x \sqsubseteq y \sqsubseteq z$. The set $\text{ipdom}_{\sqsubseteq}\left(n\right)$ is defined by

$$\text{ipdom}_{\sqsubseteq}\left(n\right) = \left\{ m \;\middle|\; \begin{array}{c} m\ 1\text{-}\sqsubseteq\ n \\ \forall m' \in N.\ m'\ 1\text{-}\sqsubseteq\ n \implies m' \sqsubseteq m \end{array} \right\}$$

In contrast to strict postdominance, $x\ 1\text{-}\sqsubseteq\ x$ might hold, namely if there is a cycle $x \sqsubseteq y \sqsubseteq x$ for $x \neq y$. $\text{ipdom}_{\sqsubseteq}\left(x\right)$ is the set of *immediate* postdominators: it contains the postdominators of x that all (other) postdominators of x postdominate.

As an example, consider the CFG in Figure 14 (left) with $\sqsubseteq_{\text{MAX}}$-postdominance. We have $\text{ipdom}_{\sqsubseteq}\left(5\right) = \{7\}$ since $7\ 1\text{-}\sqsubseteq\ 5$ and each $1\text{-}\sqsubseteq$-postdominator of 5 also postdominates 7.

```
Procedure SINK_up
workqueue ← $\mathrm{COND}_G \setminus S$ in any order
while workqueue ≠ ∅ do
    $x \leftarrow$ removeFront(workqueue)
    assert ISDOM$[x] = \bot \wedge x \notin$ PROCD
    SUCCS ← $\{ y \mid x \to y,\ y \in$ PROCD $\}$
    if SUCCS = ∅ then
        $z \leftarrow \bot$
    else
        $a \leftarrow$ lca(SUCCS)
        $z \leftarrow \begin{cases} \text{any } y \in \text{SUCCS} & \text{if } a = \bot \\ a & \text{otherwise} \end{cases}$
    end
    if $z \neq \bot$ then
        ISDOM$[x] \leftarrow z$
        processed$(x)$
    else
        pushBack(workqueue, $x$)
    end
end
```

```
Procedure SINK_down
workset ← $\{ n \mid n \in \mathrm{COND}_G \setminus S,$ ISDOM$[n] \neq \bot \}$
while workset ≠ ∅ do
    $x \leftarrow$ removeMin(workset)
    $a \leftarrow$ lca$(\{ y \mid x \to_G y \})$
    $z \leftarrow \begin{cases} \bot & \text{if } a = \bot \\ s_i & \text{if } a \in S_i \\ a & \text{otherwise} \end{cases}$
    assert ISDOM$[x] = \bot \Rightarrow z = \bot$
    if $z \neq$ ISDOM$[x]$ then
        workset ← workset ∪
            $\{n \in \mathrm{COND}_G \setminus S \mid \exists n \to y.\ x <^* y\}$
        ISDOM$[x] \leftarrow z$
    end
end
```

Fig. 13. Upward and downward iteration for algorithm 3

.

```
Input  : A transitive reduction $<$ of $\sqsubseteq$
Input  : A map SCC from nodes $x$ to the strongly connected component $c$ of $<$ s.t $x \in c$
Input  : A topological sorting sccs of all strongly connected components of $<$.
Output : $\mathrm{pdf}_{\sqsubseteq}$
for scc ∈ sccs do
    local ← $\{y \mid x \in \text{scc},\ y \to x,\ \underbrace{\neg \exists x' \in \text{scc}.\ x' < y}_{y \in \text{scc}_<}\}$
    up    ← $\{y \mid \underbrace{x \in \text{scc},\ x < z}_{z \in \text{scc}_<},\ y \in \text{PDF}[z],\ \underbrace{\neg \exists x' \in \text{scc}.\ x' < y}_{y \in \text{scc}_<}\}$
    for $x$ ∈ scc do PDF$[x]$ ← local ∪ up
end
```

Algorithm 4: Computation of $\mathrm{pdf}_{\sqsubseteq}$

$8 \notin \mathrm{ipdom}_{\sqsubseteq}(5)$ because 7 1-$\sqsubseteq$ 5 but not $7 \sqsubseteq 8$. For the cycle of 8 and 9, each of those 1-$\sqsubseteq$ -postdominates itself and the other one, so we have $\mathrm{ipdom}_{\sqsubseteq}(8) = \mathrm{ipdom}_{\sqsubseteq}(9) = \{8, 9\}$.

Next we need a generalized notion of Cytron's postdominance frontiers. Intuitively, the postdominance frontier contains all nodes that are one step away from having x as a postdominator.

Definition 5.3 ($\sqsubseteq$-Postdominance Frontiers)**.**

$$\mathrm{pdf}_{\sqsubseteq}(x) = \left\{ y \;\middle|\; \text{for some } s \text{ s.t. } y \to s : \begin{array}{r} \neg\, x \text{ 1-}\sqsubseteq y \\ x \sqsubseteq s \end{array} \right\}$$

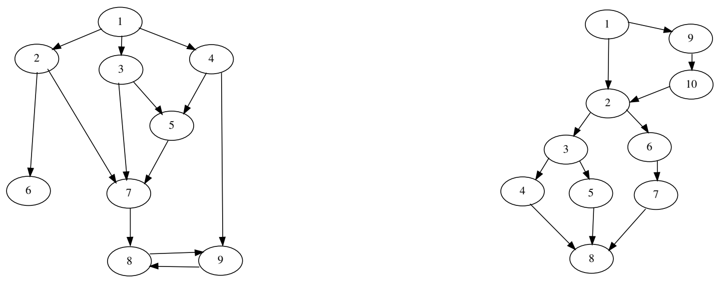


Fig. 14. Two example CFGs

Consider again Figure 14 (left) with $\sqsubseteq_{\text{MAX}}$-postdominance. We have $\text{pdf}_{\sqsubseteq}(5) = \{3, 4\}$, since 5 neither postdominates 3 or 4, but postdominates a successor of those nodes (namely 5 itself). $1 \notin \text{pdf}_{\sqsubseteq}(5)$ since 5 postdominates no successor of 1. For the node 7 we have $\text{pdf}_{\sqsubseteq}(7) = \{1, 2, 4\}$. Note that $3 \notin \text{pdf}_{\sqsubseteq}(7)$ since 7 postdominates 3.

The following lemma generalizes Cytron's insight that CD is essentially the same as postdominance frontiers:

Lemma 5.1. For $n \neq m$, we have

$$n \to_{\text{ntscd}} m \iff n \in \text{pdf}_{\sqsubseteq_{\text{MAX}}}(m)$$

and

$$n \to_{\text{nticd}} m \iff n \in \text{pdf}_{\sqsubseteq_{\text{SINK}}}(m)$$

Due to this lemma, we easily obtain $\to_{\text{ntscd}}$ and $\to_{\text{nticd}}$ once we have an algorithm for $\text{pdf}_{\sqsubseteq}$. For the latter, we – following Cytron – partition $\text{pdf}_{\sqsubseteq}(x)$ into two parts: those y contributed *locally*, and those y contributed by nodes z which are immediately $\sqsubseteq$-postdominated by x (implying $x \sqsubseteq z$). Informally, the local part $\text{pdf}^{\text{local}}_{\sqsubseteq}(x)$ of $\text{pdf}_{\sqsubseteq}(x)$ comprises all nodes from which one can get to x in one step, but which do not have x as a postdominator. On the other hand, if $y \in \text{pdf}_{\sqsubseteq}(z)$ and $\text{ipdom}_{\sqsubseteq}(z)$ is not the join point of all of y's branching, then y is in the "upper" part $\text{pdf}^{\text{up}}_{\sqsubseteq}(z)$. This is formalized in

Definition 5.4 ($\sqsubseteq$-Postdominance Frontiers: *local* and *up* part)**.**

$$\text{pdf}^{\text{local}}_{\sqsubseteq}(x) = \{y \mid \neg\ x\ 1\text{-}\sqsubseteq\ y,\ \ y \to x\}$$

$$\text{pdf}^{\text{up}}_{\sqsubseteq}(z) = \left\{y \in \text{pdf}_{\sqsubseteq}(z) \;\middle|\; \forall x \in \text{ipdom}_{\sqsubseteq}(z)\,.\ \neg\ x\ 1\text{-}\sqsubseteq\ y\right\}$$

Under suitable conditions, $\text{pdf}^{\text{up}}_{\sqsubseteq}$ and $\text{pdf}^{\text{local}}_{\sqsubseteq}$ indeed partition $\text{pdf}_{\sqsubseteq}$. This is made precise in the following

Observation 5.3. Let $\sqsubseteq$ be transitive and reflexive. Also, identify $\text{ipdom}_{\sqsubseteq}$ with the relation $\left\{(x,z) \;\middle|\; x \in \text{ipdom}_{\sqsubseteq}(z)\right\}$, and assume $\text{ipdom}^{*}_{\sqsubseteq} = \sqsubseteq$. Then

$$\text{pdf}_{\sqsubseteq}(x) = \text{pdf}^{\text{local}}_{\sqsubseteq}(x) \quad \cup \bigcup_{\left\{z \,\middle|\, x \in \text{ipdom}_{\sqsubseteq}(z)\right\}} \text{pdf}^{\text{up}}_{\sqsubseteq}(z)$$

Fortunately, $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ are reflexive and transitive (but, as explained, not antisymmetric); thus the partitioning can be applied. For an example, consider again Figure 14 (left) with $\sqsubseteq_{\text{MAX}}$-postdominance. We have $\text{pdf}^{\text{local}}_{\sqsubseteq}(5) = \{3, 4\}$. Since 5 only postdominates itself trivially, we have $5 \in \text{ipdom}_{\sqsubseteq}(z)$ for no node z, and observation 5.3 indeed gives $\text{pdf}_{\sqsubseteq}(5) = \{3, 4\}$. We have $\text{pdf}^{\text{local}}_{\sqsubseteq}(7) = \{2\}$. Since we have $\{z \mid 7 \in \text{ipdom}_{\sqsubseteq}(z)\} = \{3, 5\}$, we need to calculate $\text{pdf}^{\text{up}}_{\sqsubseteq}(3)$ and $\text{pdf}^{\text{up}}_{\sqsubseteq}(5)$. For 5, we have already seen $\text{pdf}_{\sqsubseteq}(5) = \{3, 4\}$. But $\text{pdf}^{\text{up}}_{\sqsubseteq}(5)$ contains only node 4, since 7 actually postdominates 3! Since $\text{pdf}^{\text{up}}_{\sqsubseteq}(3) = \{1\}$, observation 5.3 results in $\text{pdf}_{\sqsubseteq}(7) = \{1, 2, 4\}$, as expected.

The next definition provides properties which will enable a fixpoint computation of $\text{pdf}^{\text{local}}_{\sqsubseteq}(x)$ and $\text{pdf}^{\text{up}}_{\sqsubseteq}(z)$.

Definition 5.5. $\sqsubseteq$ is closed under $\rightarrow$, if it admits the rules

$$\frac{y \rightarrow x \quad x' \sqsubseteq y \quad x' \neq y}{x' \sqsubseteq x}\ \text{CL}^{\rightarrow}$$

$\sqsubseteq$ lacks joins if it admits the rules

$$\frac{\begin{array}{ll} x \in \text{ipdom}_{\sqsubseteq}(v) & v \sqsubseteq s \\ x \in \text{ipdom}_{\sqsubseteq}(z) & z \sqsubseteq s \end{array} \quad z \neq v}{v \in \text{ipdom}_{\sqsubseteq}(z) \ \vee z \in \text{ipdom}_{\sqsubseteq}(v)}\ \text{NoJoin}$$

Informally, the premise of the last rule is "split" at s (into v and z), and joined at x. The conclusion demands that this cannot happen unless v and z are immediate neighbours.

Lemma 5.2. Both $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ are closed under $\rightarrow$, and lack joins.

As promised, the following theorems provide, under the "lacks join" assumption for $\sqsubseteq$, simplified formulae for $\text{pdf}^{\text{local}}_{\sqsubseteq}(x)$ and $\text{pdf}^{\text{up}}_{\sqsubseteq}(z)$.

Observation 5.4. Let $\sqsubseteq$ be transitive, and closed under $\rightarrow$. Then

$$\text{pdf}^{\text{local}}_{\sqsubseteq}(x) = \{y \mid \neg\, x \in \text{ipdom}_{\sqsubseteq}(y)\ ,\ y \rightarrow x\}$$

Observation 5.5. Let $\sqsubseteq$ be transitive, reflexive, lacking joins, and closed under $\rightarrow$. Also assume $\text{ipdom}^{*}_{\sqsubseteq} = \sqsubseteq$. Then, given some z with $x \in \text{ipdom}_{\sqsubseteq}(z)$

$$\text{pdf}^{\text{up}}_{\sqsubseteq}(z) = \{y \in \text{pdf}_{\sqsubseteq}(z) \mid \neg\, x \in \text{ipdom}_{\sqsubseteq}(y)\}$$

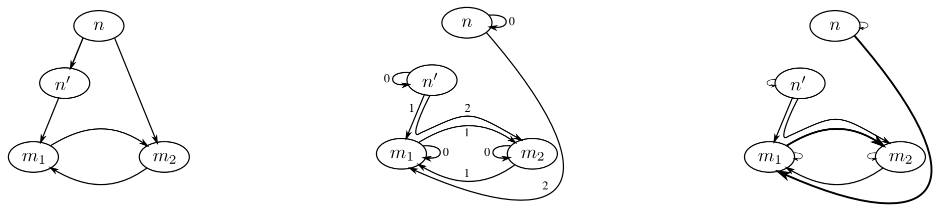
As both $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$ satisfy the assumptions of the last theorems, these theorems immediately lead to an efficient rule system for computing $\text{pdf}_{\sqsubseteq}(x)$. The first rule initializes $\text{pdf}_{\sqsubseteq}(x)$ to its "local" part; the second rule applies the formula for the "upper" part, until a fixpoint is reached. Of course, $\text{ipdom}_{\sqsubseteq}$ must be computed beforehand.

Definition 5.6. The monotone rule system for computing $\text{pdf}_{\sqsubseteq}(x)$ is given by

$$\frac{x \notin \text{ipdom}_{\sqsubseteq}(y) \quad y \rightarrow x}{y \in \text{pdf}_{\sqsubseteq}(x)} \qquad \frac{x \notin \text{ipdom}_{\sqsubseteq}(y) \quad x \in \text{ipdom}_{\sqsubseteq}(z) \quad y \in \text{pdf}_{\sqsubseteq}(z)}{y \in \text{pdf}_{\sqsubseteq}(x)}$$

The smallest fixpoint of this rule system can be computed by a standard worklist algorithm. Additionally, we can exploit transitive reductions. Given any transitive reduction $<$ of $\sqsubseteq$,

(1) compute the strongly connected components sccs of the graph $(N, <)$, in a corresponding topological order. These can either be provided by the algorithm computing $<$, or by Tarjan's algorithm [32].



(a) A CFG G (b) $\sqsubseteq^k_{\text{TIME}}$ edges reversed (c) $\sqsubseteq_{\text{TIME}}$ edges reversed

Fig. 15. An irreducible graph with *intransitive* $\sqsubseteq_{\text{TIME}}$

(2) compute $\text{pdf}_{\sqsubseteq}$ by traversing the condensed graph in that order *once*.

This concludes the algorithm for generalized postdominance frontiers $\text{pdf}_{\sqsubseteq}(x)$; and thus for $\to_{\text{ntscd}}$ and $\to_{\text{nticd}}$. For the actual computation, we propose the following optimization: By *precomputing* the set $\mathsf{scc}_< = \{ y \mid \exists x' \in \mathsf{scc}.\ x' < y \}$ for each scc, we can use this for both the tests on y, and for enumerating z.

To illustrate the fixpoint iteration for $\text{pdf}_{\sqsubseteq}(x)$, consider once more the CFG in Figure 14 (left). The "local" rule gives us e.g. $1 \in \text{pdf}_{\sqsubseteq}(3)$, $3 \in \text{pdf}_{\sqsubseteq}(5)$, $4 \in \text{pdf}_{\sqsubseteq}(5)$ and $2 \in \text{pdf}_{\sqsubseteq}(7)$. With the "up" rule we can now get $4 \in \text{pdf}_{\sqsubseteq}(7)$ by instantiating the rule with $x = 7$, $y = 4$ and $z = 5$. Note that we indeed have shown $4 \in \text{pdf}_{\sqsubseteq}(5)$ earlier and we have $7 \notin \text{ipdom}_{\sqsubseteq}(4)$ as well as $7 \in \text{ipdom}_{\sqsubseteq}(5)$. In contrast, if we try to use $3 \in \text{pdf}_{\sqsubseteq}(5)$ to show $3 \in \text{pdf}_{\sqsubseteq}(7)$ (which is false), $7 \notin \text{ipdom}_{\sqsubseteq}(3)$ would have to hold. But $7 \in \text{ipdom}_{\sqsubseteq}(3)$, so the right rule is not applicable, and we are prevented from showing $3 \in \text{pdf}_{\sqsubseteq}(7)$.

5.3 Timing Sensitive Postdominance Frontiers

In order to develop efficient algorithms for the computation of timing sensitive postdominance $\sqsubseteq_{\text{TIME}}$ and timing sensitive control-dependence $\to_{\text{tscd}}$, let us first recall that our algorithms for $\sqsubseteq_{\text{MAX}}$ and $\to$ntscd rely on the fact that $\sqsubseteq_{\text{MAX}}$ is *transitive*:

(1) Transitivity of $\sqsubseteq_{\text{MAX}}$ allows us to efficiently compute and represent $\sqsubseteq_{\text{MAX}}$ in form of its transitive reduction $<_{\text{MAX}}$. Here, $<_{\text{MAX}}$ turned out to be a pseudo-forest.

(2) Transitivity of $\sqsubseteq_{\text{MAX}}$, and the fact that

$$\text{ipdom}^*_{\sqsubseteq_{\text{MAX}}} = \sqsubseteq_{\text{MAX}}$$

allows us to use algorithm 4 to efficiently compute $\to_{\text{ntscd}}$ via $\text{pdf}_{\sqsubseteq_{\text{MAX}}}$.

Disregarding for now that $\to_{\text{tscd}}$ is defined in terms of the *ternary* relation $n \sqsubseteq^k_{\text{TIME}} m$, and not in terms of its binary "$\exists k$. -closure" $n \sqsubseteq_{\text{TIME}} m$, let us investigate first if $n \sqsubseteq_{\text{TIME}} m$ is – in general – transitive. Consider the (irreducible) CFG in Figure 15a. Here, every maximal path starting in n first reaches m_1 after two steps, hence $m_1 \sqsubseteq_{\text{TIME}} n$. Also, every maximal path starting in m_1 first reaches m_2 after one step, hence $m_2 \sqsubseteq_{\text{TIME}} m_1$. But it is for *no* number k of steps the case that $m_2 \sqsubseteq^k_{\text{TIME}} n$, hence: $\neg\ m_1 \sqsubseteq_{\text{TIME}} n$. In summary, $\sqsubseteq_{\text{TIME}}$ is *not* transitive.

Fortunately, situations as in Figure 15 are the only ones in which $\sqsubseteq_{\text{TIME}}$ is not transitive:

Theorem 5.1. Let G be any *reducible* CFG. Then $\sqsubseteq_{\text{TIME}}$ is transitive.

Theorem 5.2. Let G be any CFG with unique exit node n_x. Then $\sqsubseteq_{\text{TIME}}$ is transitive.

In practice, many programs have reducible CFGs *or* a unique exit; then $\sqsubseteq_{\text{TIME}}$ is transitive by the above two theorems. Whenever $\sqsubseteq_{\text{TIME}}$ is transitive, we can use algorithm 4 to compute $\rightarrow_{\text{tscd}}$. And if not, in [16] we present an algorithm for $\rightarrow_{\text{tscd}}$ which works even if $\sqsubseteq_{\text{TIME}}$ is not transitive. But it is much more complex and thus not described in this article. Note that even our transitive "restriction" is more general than the restriction to *structured* CFGs which is often required in literature on timing leaks, such as in e.g. [1, 17, 28].

Still, even under the $\sqsubseteq_{\text{TIME}}$ transitivity assumption, we are not done. Compared to the above $\sqsubseteq_{\text{MAX}}$ algorithm, we must deal with the ternary $n \sqsubseteq^k_{\text{TIME}} m$ instead of the binary $\sqsubseteq_{\text{MAX}}$. To this end, remember that for $m \neq n$,

$$n \in \text{pdf}_{\sqsubseteq_{\text{MAX}}}(m) \;\Leftrightarrow\; n \rightarrow_{\text{ntscd}} m$$

To obtain the analogous result for $\rightarrow_{\text{tscd}}$, we first need to "conservatively" redefine the notion $\text{pdf}_{\sqsubseteq}$ of $\sqsubseteq$-postdominance in order to obtain a notion appropriate for non-transitive relations $\sqsubseteq$. Remember that in definition 5.3, we defined for any binary relation $\sqsubseteq$:

$$\text{pdf}_{\sqsubseteq}(m) = \left\{ n \;\middle|\; \text{for some } n' \text{ s.t. } n \rightarrow_G n' : \begin{array}{r} \neg\, m \text{ 1-}\sqsubseteq n \\ m \sqsubseteq n' \end{array} \right\}$$

Syntactically, we will stick with this definition, but will modify the notion of 1-$\sqsubseteq$-postdominance. The new definition is

Definition 5.7 (1-$\sqsubseteq$-Postdominance, redefinition)**.** Given a relation $\sqsubseteq \;\subseteq N \times N$, a node $x \in N$ is said to 1-$\sqsubseteq$-postdominate z if $x \sqsubseteq z$ and there exists some node $y \neq x$ such that

$$x \sqsubseteq y \sqsubseteq z$$

The only change is the new requirement $x \sqsubseteq z$, which of course was redundant up to this section, since any relation $\sqsubseteq$ we considered (i.e.: $\sqsubseteq_{\text{POST}}$, $\sqsubseteq_{\text{MAX}}$ and $\sqsubseteq_{\text{SINK}}$) was transitive. Implicitly, this change also affects immediate $\sqsubseteq$-postdominance $\text{ipdom}_{\sqsubseteq}$ – see definition 5.2.

Theorem 5.3. Let $n \neq m \in N$. Then

$$n \in \text{pdf}_{\sqsubseteq_{\text{TIME}}}(m) \;\Leftrightarrow\; n \rightarrow_{\text{tscd}} m$$

Theorem 5.3 holds for *arbitrary* graphs, and establishes that indeed, timing sensitive postdominance frontiers are essentially timing sensitive control dependence.

But in order to use the generalized postdominance frontiers algorithm from subsection 5.2 at least for transitive $\sqsubseteq_{\text{TIME}}$, we also need the two other two requirements of that algorithm. These two do, indeed, hold even for *arbitrary* graphs:

Observation 5.6. Let $\sqsubseteq \;=\; \sqsubseteq_{\text{TIME}}$. Then $\sqsubseteq$ is closed under $\rightarrow_G$, and

$$\text{pdf}^{\text{local}}_{\sqsubseteq}(x) = \left\{ y \;\middle|\; \begin{array}{r} \neg\, x \in \text{ipdom}_{\sqsubseteq}(y) \\ y \rightarrow x \end{array} \right\}$$

Observation 5.7. Let $\sqsubseteq \;=\; \sqsubseteq_{\text{TIME}}$. Then $\sqsubseteq$ lacks joins and is closed under $\rightarrow_G$, and given some z with $x \in \text{ipdom}_{\sqsubseteq}(z)$:

$$\text{pdf}^{\text{up}}_{\sqsubseteq}(z, x) = \left\{ y \in \text{pdf}_{\sqsubseteq}(z) \;\middle|\; \neg\, x \in \text{ipdom}_{\sqsubseteq}(y) \right\}$$

All that is required now is an algorithm to compute $\sqsubseteq_{\text{TIME}}$. For graphs that are reducible, or have a unique exit node, this can be done by modifying algorithm 3 to work on $\mathbb{N}$-labeled

```
Input: A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t.
        $\mathsf{IDOM}\left[n\right] = \left(m, k\right)$ iff $m <^k n$
Input: Numbers $\mathsf{k}^{\mathsf{n}}_0$, $\mathsf{k}^{\mathsf{m}}_0 \in \mathbb{N}$ and nodes $\mathsf{n}_0$, $\mathsf{m}_0$
Output: $\mathsf{lca}_<\left(\left(\mathsf{n}_0, \mathsf{k}^{\mathsf{n}}_0\right), \left(\mathsf{m}_0, \mathsf{k}^{\mathsf{m}}_0\right)\right)$ if it exists, or $\perp$ otherwise.
return $\mathsf{lca}\left(\left(\mathsf{n}_0, \mathsf{k}^{\mathsf{n}}_0, \left[\mathsf{n}_0 \mapsto \mathsf{k}^{\mathsf{n}}_0\right]\right),\ \left(\mathsf{m}_0, \mathsf{k}^{\mathsf{m}}_0, \left[\mathsf{m}_0 \mapsto \mathsf{k}^{\mathsf{m}}_0\right]\right)\right)$
Function $\mathsf{lca}\left(\pi_{\mathsf{n}},\ \pi_{\mathsf{m}}\right)$
    Input: A cycle free $<$-path $\pi_{\mathsf{n}} = \mathsf{n}_0, \ldots, \mathsf{n}$ ending in $\mathsf{n}$, represented by a tuple $\left(\mathsf{n}, \mathsf{k}^{\mathsf{n}}, \mathsf{KS}_{\mathsf{n}}\right)$
           where $\mathsf{KS}_{\mathsf{n}}$ is a map on the nodes $n$ appearing in $\pi_{\mathsf{n}}$ s.t. $\mathsf{k}^{\mathsf{n}} = \mathsf{KS}_{\mathsf{n}}\left[\mathsf{n}\right]$ and for any
           such $n$: $\mathsf{KS}_{\mathsf{n}}\left[n\right] = \mathsf{k}^{\mathsf{n}}_0 + \sum_i k_i$ where $n <^{k_c} \ldots <^{k_1} \mathsf{n}_0$ in $\pi_{\mathsf{n}}$
    Input: A $<$-path $\pi_{\mathsf{m}} = \mathsf{m}_0, \ldots, \mathsf{m}$ likewise
    if $\mathsf{k}^{\mathsf{n}} > \mathsf{k}^{\mathsf{m}}$ then return $\mathsf{lca}\left(\pi_{\mathsf{m}},\ \pi_{\mathsf{n}}\right)$
    if $\mathsf{n} \in \pi_{\mathsf{m}} \ \wedge\ \mathsf{k}^{\mathsf{n}} = \mathsf{KS}_{\mathsf{m}}\left[\mathsf{n}\right]$ then return $\left(\mathsf{n}, \mathsf{k}^{\mathsf{n}}\right)$
    if $\mathsf{n} \in \pi_{\mathsf{m}} \ \wedge\ \mathsf{k}^{\mathsf{n}} \neq \mathsf{KS}_{\mathsf{m}}\left[\mathsf{n}\right]$ then return $\perp$
    switch $\mathsf{IDOM}[\mathsf{n}]$ do
        case $\perp$ do return $\perp$
        case $\left(\mathsf{n}', \mathsf{k}^{\mathsf{n}'}\right)$ do
            if $\mathsf{n}' \in \pi_{\mathsf{n}}$ then return $\perp$
            $\mathsf{KS}_{\mathsf{n}}\left[\mathsf{n}'\right] \leftarrow \mathsf{k}^{\mathsf{n}} + \mathsf{k}^{\mathsf{n}'}$
            return $\mathsf{lca}\left(\left(\mathsf{n}', \mathsf{k}^{\mathsf{n}} + \mathsf{k}^{\mathsf{n}'}, \mathsf{KS}_{\mathsf{n}}\right),\ \pi_{\mathsf{m}}\right)$
        end
    end
end
```

Algorithm 5: A timing sensitive *least common ancestor* algorithm for graphs with transitive $\sqsubseteq_{\text{TIME}}$.

pseudo-forests, i.e.: pseudo forests $<$ with edges $n <^k m$ indicating that m must first be reached from n after $k \in \mathbb{N}$ steps. The result is a $\mathbb{N}$-labeled pseudo-forest $<$ with

$$m \sqsubseteq_{\text{TIME}} n \iff \exists k_1, \ldots, k_c.\ m <^{k_c} \ldots <^{k_1} n$$

for some number $c \geq 0$ of edges in $<$. One possible implementation of the required least common ancestor computation in $\mathbb{N}$-labeled pseudo-forests is shown in algorithm 5.

For an example, consider Figure 14 (right). Before the first call to `lca`, `IDOM` contains only trivial relations for nodes with exactly one successor, e.g. `IDOM`$[4] = (8, 1)$. To calculate `IDOM`$[2]$, we need to call `lca` with its successors, namely `lca`$((3, 1),\ (6, 1))$. But there, we find that `IDOM`$[3]$ is still empty, so the call returns $\perp$.

When calculating `IDOM`$[3]$, we call `lca`$((4, 1),\ (5, 1))$. We find that `IDOM`$[4] = (8, 1)$, so we extend this $<$-path and call `lca`$([(4, 1), (8, 2)],\ (5, 1))$. There, since the left path is now longer, we swap the arguments and call `lca`$((5, 1),\ [(4, 1), (8, 2)])$. Now, we find that `IDOM`$[5] = (8, 1)$, so we extend this path and call `lca`$([(5, 1), (8, 2)],\ [(4, 1), (8, 2)])$. Now, since the final element of the left path, namely 8, is also contained in the right one with the same distance of 2, we finally can return $(8, 2)$ as the `lca` and update `IDOM`$[3] = (8, 2)$.

Now we can analyse `IDOM`$[2]$ again. Since `IDOM`$[3]$ has now an entry, we can extend the path $(3, 1)$ to $[(3, 1), (8, 3)]$. After extending $(6, 1)$ to $[(6, 1), (7, 2)]$ and then $[(6, 1), (7, 2), (8, 3)]$, both paths contain 8 with the same distance 3, so we update `IDOM`$[3]$ to $(8, 3)$.

On the contrary, if we try to calculate `IDOM`$[1]$ and call `lca`$((2, 1),\ (9, 1))$, the left path get extended to $[(2, 1), (8, 4)]$ and the right path to $[(9, 1), (10, 2), (2, 3)]$. Now, both paths contain the same node 2, but with different distances 1 and 3. Therefore, the `lca` is $\perp$.

6 MEASUREMENTS

We evaluated the performance of our algorithms on a) control flow graphs of Java methods, as generated by the JOANA system for various third party Java programs; b) randomly generated graphs $G = \big(N, E\big)$ usually with $|E| = 2\,|N|$, as generated by the standard generator from the JGraphT [26] library. In some cases, we additionally use ladder graphs[15], which are used to represent bad case behaviour.

All benchmarks in this section were made on a desktop computer with an Intel i7-6700 CPU at 3.40GHz, and 64 GB RAM. We implemented the algorithms in Java, using OpenJDK Java 9 VM. All benchmarks were run using the Java Microbenchmark Harness JMH [9].

Unless explicitly stated otherwise, all data points represent the average over $n + 1$ runs of the benchmark, where n is at least the number of runs which can be finished within 1 second. For example, the data point at $|N| = 21076$, time = 18ms in Figure 16a stands for the average of at least ≈ 50 runs of the benchmark that finished within 1 second. On the other hand, the data point in at $|N| = 65000$, time = 88s in Figure 16c results from only one run of the benchmark.

The purpose of these benchmarks is to give a general idea of the scalability of the algorithms. For example, the benchmarks in the upper left and upper right of Figure 18 suggest that our new algorithm for the computation of nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ appears to scale almost linearly for "average" CFGs, while Ranganath's original algorithm [29] clearly grows super-linearly for such graphs. The benchmarks can be summarized as follows:

(1) For "average" CFGs, our algorithms for $\rightarrow_{\text{ntscd}}$, $\rightarrow_{\text{nticd}}$, and $\rightarrow_{\text{tscd}}$ offer performance "almost linear" in the size of the graph.
(2) But for "bad case" CFGs, some algorithms perform decidedly super-linear, and become impractical for very large such graphs.

6.1 Nontermination Sensitive Postdominance

Algorithm 2 computes maximal path postdominance $\sqsubseteq_{\text{MAX}}$, represented as a pseudo-forest $<_{\text{MAX}}$. This algorithm requires the computation of least common ancestors $\text{lca}_<$ in pseudo-forests $<$, for which we use algorithm 1.

Algorithm 2 repeatedly iterates in a fixed node order. Alternatively, one can implement a chaotic iteration, by reinserting into a workset those nodes affected by modification to the pseudo-forest. Both these variants do not specify an iteration order (e.g.: Algorithm 2 does not specify the initial order of nodes in the workqueue). By default, the implementation orders the nodes reversed-topologically (as computed by an implementation of Kosaraju's Algorithm for strongly connected components, with nodes in the same strongly connected component ordered arbitrarily).

For Java CFG and randomly generated graphs (neither necessarily with unique exit node), the chaotic iteration (✚) and Algorithm 2 (▼) behave similarly (Figure 16a and Figure 16b). Ladder graphs expose non-linear *bad-case* behavior (Figure 16c). This is even more pronounced when we deliberately choose a bad iteration order (Figure 16d).

6.2 Nontermination Insensitive Postdominance

Algorithm 3 computes sink path postdominance $\sqsubseteq_{\text{SINK}}$, represented as a pseudo-forest $<_{\text{SINK}}$. Just as before, it uses algorithm 1 for the computation of least common ancestors $\text{lca}_<$.

[15]Ladder graphs consist of two rising chains, one-to-one connected at every node. Just like a ladder.

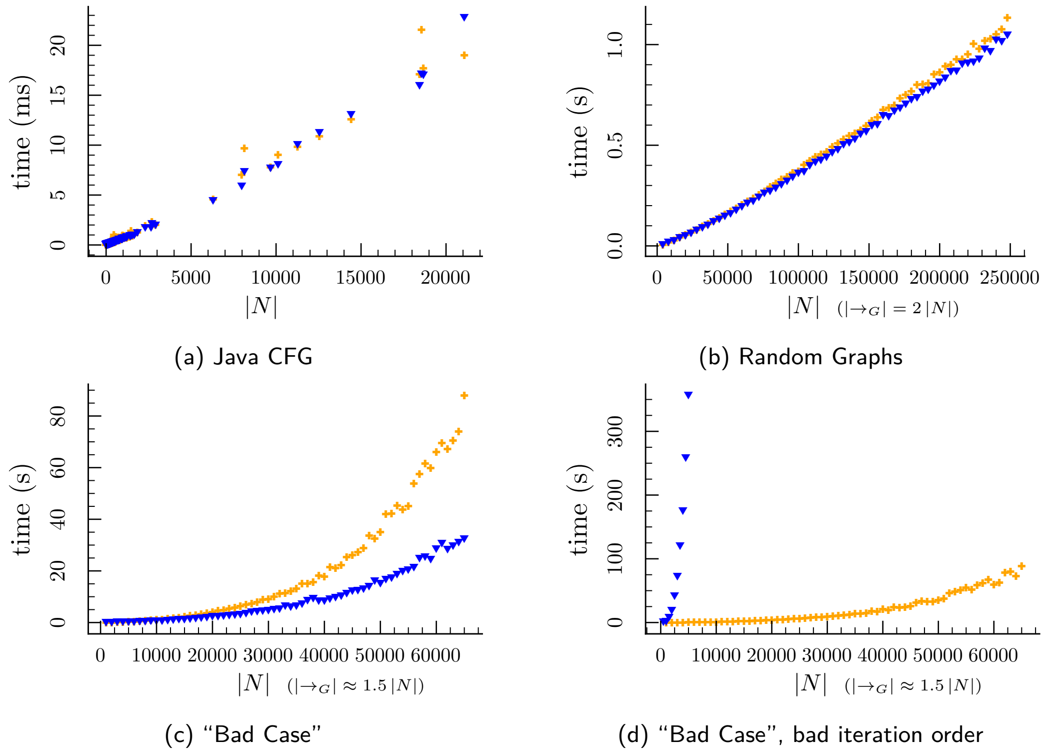


Fig. 16. Computation of $<_{\text{MAX}}$. The orange line shows chaotic iteration performance, the blue line shows algorithm 2.

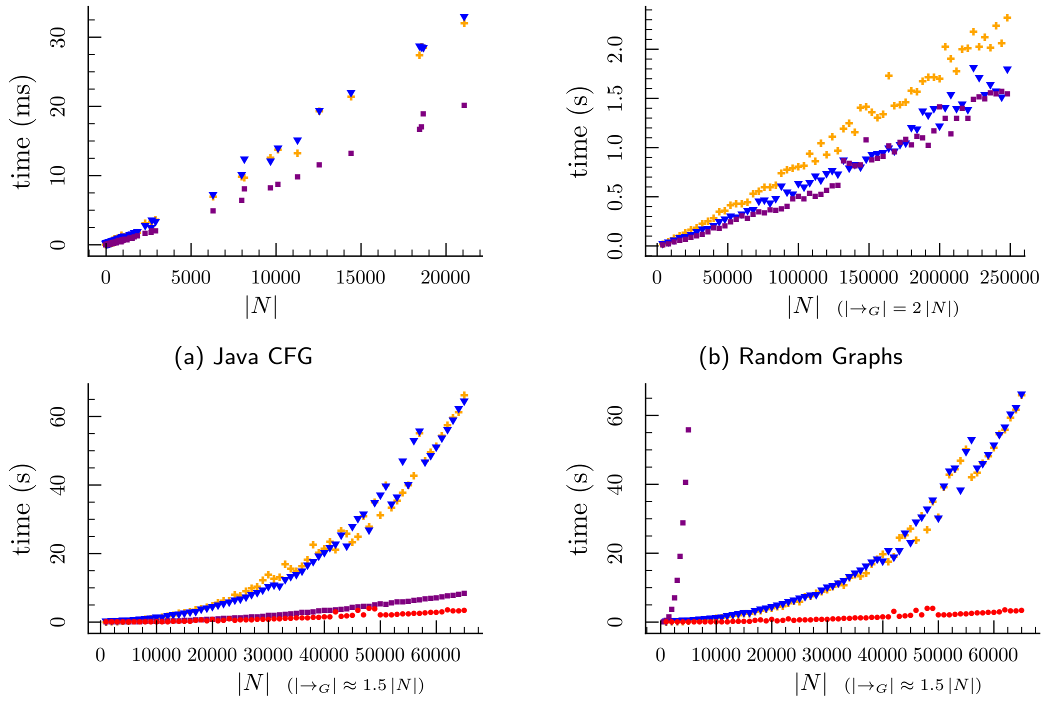
Algorithm 3 implements chaotic iteration. We also implemented a variant of Algorithm 3 in which the downward fixed point phase repeatedly iterates a workqueue of nodes in a fixed node order. The implementations order the nodes reversed-topologically. Unlike before, this ordering does not require an additional step, since the strongly connected component computation it can be obtained from is necessary anyway, in order to find *control sinks*.

Instead of computing least common ancestors $\text{lca}_<$ by chasing (pseudo-tree) pointers, it can also be computed by comparison of postorder numbers, as in [8].

For Java CFGs (Figure 17a) the fixed-iteration order variant of Algorithm 3 (▼) performs on par with the Algorithm 3 as stated (✚). For randomly generated graphs (Figure 17b) the variant (▼) appears to perform a bit better than the original (✚) for very large graphs, roughly on-par with the implementation based on postorder numbers (■).

Using reversed-topological iteration order, ladder graphs (Figure 17c) expose non-linear *bad-case* behavior only for Algorithm 3 (✚) and its variant (▼). Even with a bad iteration order, performance for these two algorithm is not much worse (Figure 17d). On the other hand, the postorder number based implementation (■) is affected heavily by iteration order.

The ladder graphs we use are unique-exit-node ladder graphs. This also allows us to directly compare with an implementation of the algorithm by Lengauer and Tarjan [24] (●).



(a) Java CFG

(b) Random Graphs

(c) "Bad Case"

(d) "Bad Case", bad iteration order

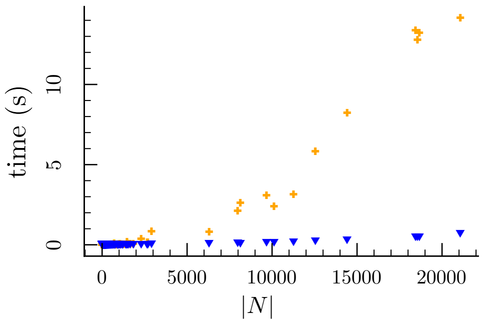
Fig. 17. Computation of $<_{\text{SINK}}$.

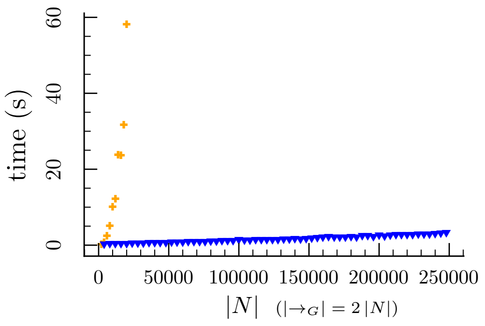
6.3 Generalized Postdominance Frontiers

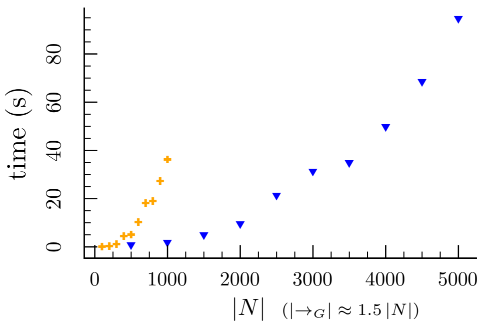
When algorithm 4 is instantiated with $<_{\text{MAX}}$, this yields an algorithm for $\rightarrow_{\text{ntscd}}$. The benchmarks for $\rightarrow_{\text{ntscd}}$ include the computation time of both algorithm 4 and $<_{\text{MAX}}$ (▼). We compare with an implementation of Ranganath's algorithm [29] (✚). For Java CFG and randomly generated graphs, the latter becomes impractical for moderately sized graphs, while algorithm 4 performs well even for very large graphs (Figure 18, upper left and right). Ladder graphs expose non-linear *bad-case* behavior even for algorithm 4 (Figure 18c). This cannot be circumvented, since in these ladder graphs, the size of the relation $\rightarrow_{\text{ntscd}}$ is quadratic in the number of nodes.

6.4 Timing Sensitive CD

Whenever $\sqsubseteq_{\text{TIME}}$ is transitive, we can use algorithm 4 to compute timing sensitive control dependence $\rightarrow_{\text{tscd}}$. We thus measure the computation time for $\rightarrow_{\text{tscd}}$ on graphs for which $\sqsubseteq_{\text{TIME}}$ is transitive. These are control flow graphs from Java programs in subfigures (a), randomly generated graphs (b), and ladder graphs (c). We use algorithm 4, and obtain a transitive reduction $<_{\text{TIME}}$ of $\sqsubseteq_{\text{TIME}}$ via the modification of algorithm 3 that uses the upwards iteration of algorithm 5. The benchmarks for $\rightarrow_{\text{tscd}}$ in Figure 19 include the computation time of all sub-algorithms (✚). Ladder graphs expose non-linear *bad-case* behavior.

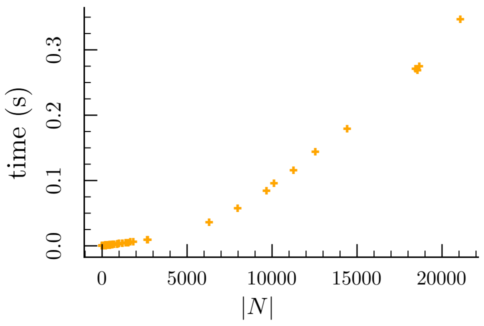


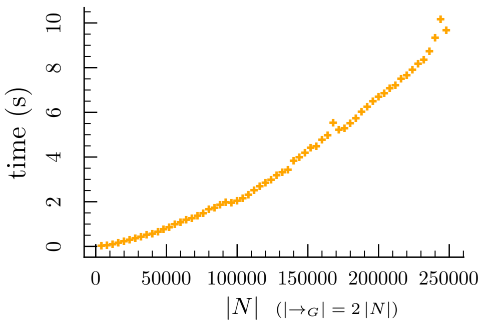


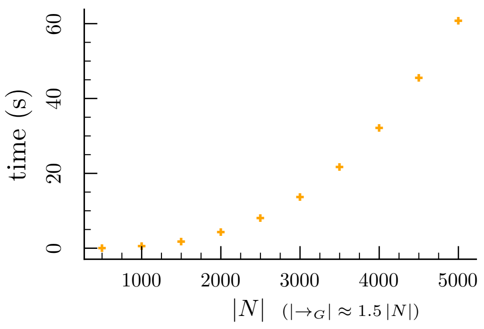


(c) “Bad Case”

Fig. 18. Computation of $\rightarrow$ntscd.

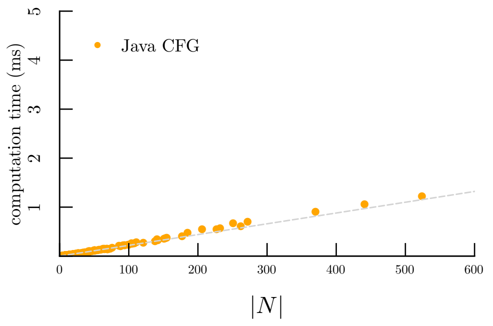






(c) “Bad Case”

Fig. 19. Computation of $\rightarrow$tscd.



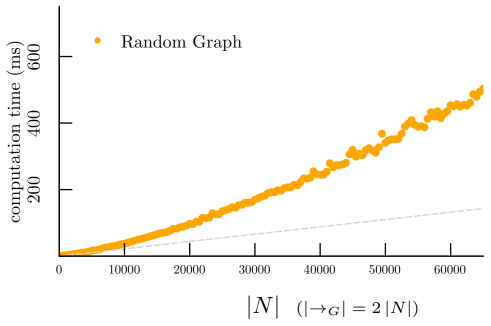


Fig. 20. Computation of $\rightarrow_{\text{nticd}}$ via algorithm 4 based on algorithm 3.

7 FUTURE WORK

This article concentrated on the definition of $\rightarrow_{\text{tscd}}$, and on efficient algorithms. Ongoing work includes

- provide Isabelle proofs for the last 7 “observations” in section 5.
- provide formal correctness proofs for the algorithms in section 5.
- implement and evaluate the $\rightarrow_{\text{tscd}}$ algorithm which can handle nontransitive $\sqsubseteq_{\text{TIME}}$, which was mentioned in section 5.3.
- provide a theoretical complexity analysis of the algorithms, and more measurements.
- transform out timing leaks as in [1], but for arbitrary CFGs (based on $\rightarrow_{\text{tscd}}$).
- apply $\rightarrow_{\text{tscd}}$ to improve IFC and probabilistic noninterference; in particular improve precision of the so-called “RLSOD” algorithm [6, 7, 12] which is used in JOANA.

Initial work on some of these topics can be found in the first author’s dissertation [16]. A long-time goal is an interprocedural, context-sensitive extension of $\rightarrow_{\text{tscd}}$.

8 CONCLUSION

Ranganath and Amtoft opened the door to control dependencies in nonterminating programs. Inspired by this work, we presented 1. new, efficient algorithms for Ranganath’s nontermination-(in)sensitive control dependencies; 2. definitions and algorithms for time-sensitive control dependencies; 3. application of the latter to timing leaks in software security. Our algorithms are based on systematic generalizations of Cytron’s postdominance frontier algorithm. Important properties of the new algorithms have been proven using the Isabelle machine prover; and their performance has been studied. We believe that *time-sensitive control dependencies* will prove useful for many applications in program analysis, code optimization, and software security.

Acknowledgements. The work described in this paper was funded by Deutsche Forschungsgemeinschaft Grant Sn11-12/3 in the scope of SPP 1496 “Reliably Secure Software”, and by BMBF Grant 01BY1172 in the scope of the Security Competence Center KASTEL.

Preliminary versions of parts of this paper have been published in the first author’s dissertation [16].

REFERENCES

[1] Johan Agat. Transforming out timing leaks. In *POPL ’00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53, New York, NY, USA, 2000. ACM.

[2] A.V. Aho, M.R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1 (2):131–137, 1972.

[3] Torben Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processing Letters*, 106(2):45 – 51, 2008.

[4] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[5] David W. Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Trans. Software Eng.*, 32(9):698–717, 2006.

[6] Simon Bischof, Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Low-deterministic security for low-nondeterministic programs. *Journal of Computer Security*, 26:335–366, 2018.

[7] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. On improvements of low-deterministic security. In *Proc. Principles of Security and Trust (POST)*, volume 9635 of *Lecture Notes in Computer Science*, pages 68–88. Springer Berlin Heidelberg, 2016.

[8] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 2001.

[9] Oracle Corporation. Code tools: jmh, 2020.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[12] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, April 2015.

[13] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Tool demonstration: JOANA. In *Proc. Principles of Security and Trust (POST)*, volume 9635 of *Lecture Notes in Computer Science*, pages 89–93. Springer Berlin Heidelberg, 2016.

[14] 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):399–422, Dec 2009.

[15] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 207–217, New York, NY, USA, 1973. ACM.

[16] Martin Hecker. *Timing Sensitive Dependency Analysis and its Application to Software Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2020.

[17] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electron. Notes Theor. Comput. Sci.*, 141(1):163–182, 2005.

[18] Susan Horwitz, Jan Prins, and Thomas W. Reps. On the adequacy of program dependence graphs for representing programs. In *POPL*, pages 146–157, 1988.

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

[20] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[21] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *32nd IEEE Symposium on Security and Privacy*, pages 413–428, 2011.

[22] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[23] Jens Krinke. Context-sensitive slicing of concurrent programs. In *Proc. FSE*, pages 178–187. ACM, 2003.

[24] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[25] Steven Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[26] Barak Naveh and Stephane Popinet. JGraphT: A java library of graph theory data structures and algorithms. https://jgrapht.org/, 2003–2019.

[27] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, Sep 1990.

[28] Willard Rafnsson, Limin Jia, and Lujo Bauer. Timing-sensitive noninterference through composition. In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust (POST)*, volume 10204 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2017.

[29] 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), August 2007.

[30] 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.

[31] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 200–214, 2000.

[32] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[33] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.