The Journal of Systems & Software 184 (2022) 111140

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Efficient computation of minimal weak and strong control closure☆

Abu Naser Masud

School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden

ARTICLE INFO

Article history:
Received 5 March 2021
Received in revised form 4 August 2021
Accepted 18 October 2021
Available online 8 November 2021

Keywords:
Control dependency
Weak control closure
Strong control closure
Program slicing
Nontermination (in)sensitive

ABSTRACT

Control dependency is a fundamental concept in many program analyses, transformation, parallelization, and compiler optimization techniques. An overwhelming number of definitions of control dependency relations are found in the literature that capture various kinds of program control flow structures. Weak and strong control closure (WCC and SCC) relations capture nontermination insensitive and sensitive control dependencies and subsume all previously defined control dependency relations. In this paper, we have shown that static dependency-based program slicing requires the repeated computation of WCC and SCC. The state-of-the-art WCC and SCC algorithm provided by Danicic et al. has the cubic and the quartic worst-case complexity in terms of the size of the control flow graph and is a major obstacle to be used in static program slicing. We have provided a simple yet efficient method to compute the minimal WCC and SCC which has the quadratic and cubic worst-case complexity and proved the correctness of our algorithms. We implemented ours and the state-of-the-art algorithms in the Clang/LLVM compiler framework and run experiments on a number of SPEC CPU 2017 benchmarks. Our WCC method performs a maximum of 23.8 times and on average 10.6 times faster than the state-of-the-art method to compute WCC. The performance curves of our WCC algorithm for practical applications are closer to the NlogN curve in the microsecond scale. Our SCC method performs a maximum of 226.86 times and on average 67.66 times faster than the state-of-the-art method to compute SCC. Evidently, we improve the practical performance of WCC and SCC computation by an order of magnitude.

1. Introduction

Control dependency is a fundamental concept in many program analyses, transformation, parallelization and compiler optimization techniques. It is used to express the relation between two program statements such that one decides whether the other statement can be executed or not. One of the key applications of control dependency is program slicing (Weiser, 1981) that transforms an original program into a sliced program with respect to a so-called slicing criterion. The slicing criterion specifies the variables at a particular program point that will affect the execution of the sliced program. All program instructions in the original program that do not affect the slicing criterion are discarded from the sliced code. Control dependency is used to identify the program instructions that indirectly affect the slicing criterion due to the execution of conditional expressions in the loops or conditional instructions.

The standard definition of control dependency provided by Ferrante et al. (1987) has been widely used for over two decades. This definition is provided at the level of the *control flow graph* (CFG) representation of a program assuming that the CFG has a unique *end* node (i.e. the program has a single exit point). Several recent articles on control dependency illustrate that this definition does not sufficiently capture the intended control dependency of programs having the modern programming language features. For instance, the *exception* or *halt* instructions cause multiple exits of the programs, or reactive systems, web services or distributed real-time systems have nonterminating program instructions without an end node. The standard definition of control dependency did not intend to handle the above systems. The possibility of having nontermination in the program code introduces two different types of control dependency relations: the *weak* and *strong* control dependencies that are nontermination insensitive and nontermination sensitive. One of the distinguishing effects between the two types of control dependencies is that an original nonterminating program remains nonterminating or may be transformed into a terminating program if the slicing method uses *strong* or *weak* control dependence respectively.

Numerous authors provided an overwhelming number of definitions of control dependencies (Weiser, 1981; Ferrante et al., 1987; Podgurski and Clarke, 1990; Bilardi and Pingali, 1996; Ranganath et al., 2007) given at the level of CFG and describe computation methods to obtain such dependencies. Danicic et al. (2011) unified all previously defined control dependence relations by

☆ Editor: W.K. Chan.

E-mail address: masud.abunaser@mdh.se.

https://doi.org/10.1016/j.jss.2021.111140

providing the definitions and theoretical insights of *weak* and *strong control-closure* (WCC and SCC) that are most generalized and capture all non-termination insensitive and nontermination sensitive control dependence relations. Thus, WCC and SCC subsume all control dependency relations found in the literature. However, Danicic et al. provided expensive algorithms to compute WCC and SCC. In particular, the algorithms for computing WCC and SCC have the cubic and quartic worst-case asymptotic complexity in terms of the size of the CFG. We have shown that static program slicing requires the repeated computation of WCC and/or SCC. The state-of-the-art WCC and SCC algorithms are not only expensive, but the use of these algorithms in client applications such as program slicing will make these applications underperforming. In other words, these applications will run slower using computationally expensive WCC and SCC algorithms even though they will obtain wider applicability than applications using the standard control dependency algorithms such as postdominator-based algorithms.

In this article, we have provided simple and efficient methods to compute WCC and SCC. We have formalized several theorems and lemmas demonstrating the soundness, minimality, and complexity of our algorithms. Our WCC and SCC algorithms have the quadratic and the cubic worst-case time complexity in terms of the size of the CFG which are improvements over the methods of Danicic et al. by an order of the size of the CFG. We implemented ours and the WCC and SCC algorithms of Danicic et al. in the Clang/LLVM compiler framework (Lattner and Adve, 2004) and performed experiments on a number of benchmarks selected from SPEC CPU 2017 (Bucek et al., 2018). Our WCC algorithm performs a maximum of 23.8 times and on average 10.6 times faster than the WCC algorithm of Danicic et al. Moreover, the practical performance of our WCC algorithm is closer to the NlogN curve. Our SCC method performs a maximum of 226.86 times and on average 67.66 times faster than the state-of-the-art method to compute SCC. Thus we improve the theoretical as well as the practical performance of WCC and SCC computation by an order of magnitude.

Note that this is an extended version of the paper entitled "Simple and Efficient Computation of Minimal Weak Control Closure" (Masud, 2020) and published at the International static analysis symposium (SAS 2020). More specifically, this article has been extended as follows:

- We have recalled the related definitions of strong control closure from Danicic et al. (2011) in Section 2.4 and provided a detailed example on the related concepts.
- We have included a new section (Section 5) with the formal development of our algorithms (Algorithms 6 and 7) to compute SCC. We have provided several lemmas as the theoretical foundation of our efficient SCC computation algorithm. We have proved that our algorithm generates the minimal SCC (Lemma 8). We have provided an incremental method to compute SCC (Algorithm 8, Algorithm 9, and Algorithm 10) which is even more efficient than Algorithm 7. We have proved the theoretical worst-case complexity of our algorithms.
- We have implemented our SCC computation algorithm in the Clang/LLVM framework and performed additional experiments comparing ours and the state-of-the-art algorithms to compute SCC. Experimental results are included in Section 6. The results show that our algorithms are outstandingly faster than the state-of-the-art algorithms.

The remainder of this paper is organized as follows. Section 2 provides some notations and backgrounds on WCC and SCC, Section 3 illustrates the changes to be performed in static program slicing due to WCC/SCC, Section 4 provides the detailed description of our WCC computation method, proves the correctness, and the worst-case time complexity of our method, Section 5 provides the detailed description of our SCC computation method, proof of correctness, and the worst-case time complexity of our method, Section 6 compares the performance of ours and the state-of-the-art WCC and SCC computation method of Danicic et al. on some practical benchmarks, Section 7 discusses the related works, and Section 8 concludes the paper.

2. Background

2.1. CFG and related concepts

We provide the following formal definition of control flow graph (CFG).

Definition 1 (*CFG*)**.** A CFG is a directed graph (N, E) where

1. N is the set of nodes that includes a *Start* node from where the execution starts, at most one *End* node where the execution terminates normally, *Cond* nodes representing boolean conditions, and *nonCond* nodes; and
2. $E \subseteq N \times N$ is the relation describing the possible flow of execution in the graph. An *End* node has no successor, a *Cond* node n has at most one *true* successor and at most one *false* successor, and all other nodes have at most one successor.

Like Danicic et al. (2011), we assume the following:

- The CFG is deterministic. So, any *Cond* node n cannot have multiple true successors and/or multiple false successors.
- We allow *Cond* nodes to have either or both of the successors missing. We may also have non-*End* nodes having no successor (i.e. out-degree zero). An execution that reaches these nodes are silently nonterminating as it is not performing any action and does not return control to the operating system.
- If the CFG G has no *End* nodes, then all executions of G are nonterminating.
- Moreover, if a program has multiple terminating exit points, nodes representing those exit points are connected to the *End* node to model those terminations. Thus, the CFG in Definition 1 is sufficiently general to model a wide-range of real-world intraprocedural programs.

The sets of successor and predecessor nodes of any CFG node n in a CFG (N, E) are denoted by $succ(n)$ and $pred(n)$ where $succ(n) = \{m : (n, m) \in E\}$ and $pred(n) = \{m : (m, n) \in E\}$.

Definition 2 (*CFG Paths*)**.** A path π is a sequence $n_1, n_2, \ldots, n_k$ of CFG nodes (denoted by $[n_1 \ldots n_k]$) such that $k \geq 1$ and $n_{i+1} \in succ(n_i)$ for all $1 \leq i \leq k-1$.

A path is *non-trivial* if it contains at least two nodes. We write $\pi - S$ to denote the set of all nodes in the path π that are not in the set S. The length of any path $[n_1 \ldots n_k]$ is $k-1$. A trivial path $[n]$ has path length 0.

Definition 3 (*Disjoint Paths*)**.** Two finite paths $[n_1 \ldots n_k]$ and $[m_1 \ldots m_l]$ such that $k, l \geq 1$ in any CFG G are disjoint paths if and only if no n_i is equal to m_j for all $1 \leq i \leq k$ and $1 \leq j \leq l$. In other words, the paths do not meet at a common vertex.

Sometimes, we shall use the phrase "two disjoint paths from n" to mean that there exist two paths $n_1 = n, \ldots, n_k$ and $n'_1 = n, \ldots, n'_l$ such that $[n_2 \ldots n_k]$ and $[n'_2 \ldots n'_l]$ are disjoint paths.

In other words, the paths are disjoint after the first common vertex. A CFG node is *final* if it is either a Cond node not having both a true and a false successor or a non-Cond node having no successor. A CFG path is *complete* if it is either an infinite path or a finite path whose last vertex is a final node.

2.2. Standard control dependency

Ferrante et al. (1987) provided the first formal definition of control dependency relation based on *postdominator* (Prosser, 1959) relation. Computing postdominator relations on a CFG G requires that G has a single End node n_e and there is a path from each node n in G to n_e. A node n *postdominates* a node m if and only if every path from m to n_e goes through n. Node n *strictly postdominates* m if n postdominates m and $n \neq m$. The standard *postdominator-based control dependency relation* can then be defined as follows:

Definition 4 (*Control Dependency Ferrante et al., 1987; Ranganath et al., 2007*)**.** Node n is *control dependent* on node m (written $m \xrightarrow{cd} n$) in the CFG G if (1) there exists a nontrivial path π in G from m to n such that every node $m' \in \pi - \{m, n\}$ is postdominated by n, and (2) m is not strictly postdominated by n.

The relation $m \xrightarrow{cd} n$ implies that there must be two branches of m such that n is always executed in one branch and may not execute in the other branch.

Example 1. The CFG in Fig. 1(b) is obtained from the source code in Fig. 1(a) according to Definition 1. This pattern of CFG is abundant in the perlbench in SPEC CPU2017 (Bucek et al., 2018). We shall use this CFG as a running example in the remainder of this article. The labeling of *true* and *false* branches of Cond nodes are omitted for simplicity. Fig. 2 presents the control dependency graph (CDG) computed from the CFG based on computing postdominator relations such that an edge (n, m) in the CDG represents the control dependency relation $n \xrightarrow{cd} m$. For example, the edge (n_{10}, n_6) in the CDG represents that statement 21 is control dependent on statement 8 in the program in Fig. 1(a).

2.3. Weak control closure

Podgurski and Clarke (1990) introduced the weak control dependence which is nontermination sensitive. A number of different nontermination sensitive and nontermination insensitive control dependency relations conservatively extending the standard relation above are defined in successive works (Amtoft, 2007; Ottenstein and Ottenstein, 1984; Pingali and Bilardi, 1997; Ranganath et al., 2007; Podgurski and Clarke, 1990). Danicic et al. (2011) unified all previous definitions and presented two generalizations called *weak* and *strong control closure* which are non-termination insensitive and non-termination sensitive. WCC and SCC capture all the existing non-termination (in)sensitive control dependency relations found in the literature. In this section, we recall some relevant definitions and terminologies of WCC from Danicic et al. (2011).

Definition 5 (N'*-Path*)**.** An N'-path is a finite path $[n_1 \dots n_k]$ in a CFG G such that $n_k \in N'$ and $n_i \notin N'$ for all $1 < i \leq k - 1$.

Note that n_1 may be in N' in the above definition. Thus, an N'-path from n ends at a node in N' and no node in this path is in N' except n_1 which may or may not be in N'.

Definition 6 (N'*-Weakly Committing Vertex*)**.** Let $G = (N, E)$ be any CFG. A node $n \in N$ is N'-weakly committing in G if all N'-paths from n have the same endpoint. In other words, there is at most one element of N' that is 'first-reachable' from n.

Definition 7 (*Weak Control Closure*)**.** Let $G = (N, E)$ be any CFG and let $N' \subseteq N$. N' is weakly control-closed in G if and only if all nodes $n \in N \setminus N'$ that are reachable from N' are N'-weakly committing in G.

We say that a set X is weakly control-closed if and only if it is the weak control closure of any set of CFG nodes. The concept of weakly deciding vertices is introduced to prove that there exists minimal and unique WCC of a set of nodes $N' \subseteq N$. Since program slicing uses control dependence relations to capture all control dependent nodes affecting the slicing criterion, using minimal WCC in program slicing gives us smaller nontermination insensitive slices.

Definition 8 (*Weakly Deciding Vertices*)**.** A node $n \in N$ is N'-weakly deciding in G if and only if there exist two finite proper N'-paths in G that both start at n and have no other common vertices. $WD_G(N')$ denotes the set of all N'-weakly deciding vertices in G.

Thus, if there exists an N'-weakly deciding vertex n, then n is not N'-weakly committing. The WCC of an arbitrary set $N' \subseteq N$ can be formally defined using weakly deciding vertices as follows:

$$WCC(N') = \{n : n \in WD_G(N'), n \text{ is reachable from } N' \text{ in } G\} \cup N'$$

Example 2. Consider the CFG in Fig. 1. Let $N' = \{n_5, n_8, n_{10}\}$. The N'-paths in this CFG include $n_9, \dots, n_5$ and $n_4, \dots, n_6, n_4, n_8$. The path n_6, n_5, n_4, n_8 is not an N'-path since $n_5 \in N'$. Nodes n_{12}, n_{13}, n_{14} and n_{15} are N'-weakly committing. However, n_9 and n_6 are not N'-weakly committing due to the N'-paths $[n_9 \dots n_{10}]$ and $[n_9 \dots n_5]$, and n_6, n_5 and n_6, n_4, n_8. Nodes n_9 and n_6 are thus N'-weakly deciding and N' is not weakly control closed. However, all N'-weakly deciding vertices n_4, n_6 and n_9 are reachable from N' and thus $N' \cup \{n_4, n_6, n_9\}$ is a weak control-closed set capturing all the relevant control dependencies of N'.

2.4. Strong control closure

In this section, we recall some relevant definitions and terminologies of SCC from Danicic et al. (2011).

Definition 9 (*Strongly Committing Vertices*)**.** Let $n \in N$ be any CFG node. n is N'-strongly committing in G iff it is N'-weakly committing in G and all complete paths in G from n contain an element of N'.

Definition 10 (N'*-Avoiding Vertices*)**.** A node $n \in N$ is N'-avoiding in G if and only if no node in N' is reachable in G from n.

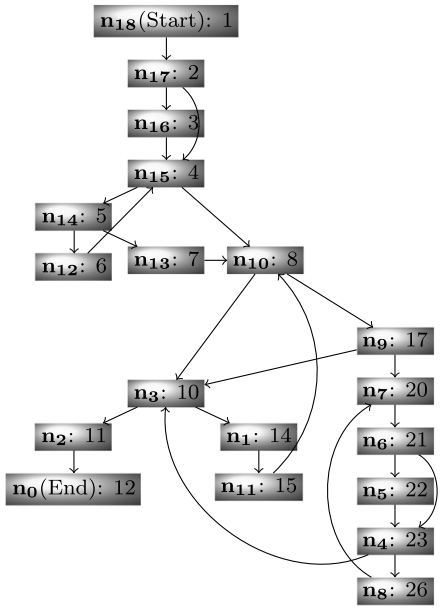
Definition 11 (*Strong Control Closure*)**.** Let N' be any subset of N. N' is strongly control-closed in G if and only if all nodes $n \in N \setminus N'$ that are reachable in G from N' are N'-strongly committing or N'-avoiding in G.

We say that a set X is strongly control-closed if and only if it is the strong control closure of any set of CFG nodes.

Example 3. Consider the CFG G in Fig. 1. Let G_1 be the CFG G after removing the nodes n_{12}, n_{13}, and n_0, and all incoming and outgoing edges from these nodes. Thus, all executions of G_1 are nonterminating. Some program analyses such as program slicing

```
1 void run(){
2   if (! in_safe )
3         in_safe =true;
4   while(load<capacity)
5         if (load<max_val)
6               load++;
7         else break;
8   while(overhead<ov_limit){
9         CRITICAL:
10        if (! in_safe ) {
11              print(Error);
12              return; }
14        overhead++;
15        loadnexec(overhead, load);
16    }
17  if (capacity<max_val)
18        goto CRITICAL;
19  else{
20      while(true){
21       if (overhead ≥ ov_limit)
22         overhead=capacity−load;
23       if (in_safe )
24         goto CRITICAL;
25       else
26        up_status(load, capacity);
27}}}
```

(a)



(b)

Fig. 1. (a) Code snippet used as a running example in which all variables are global, (b) intraprocedural CFG of (a) according to Definition 1 in which $n : l$ represents the CFG node n originated from statement l.

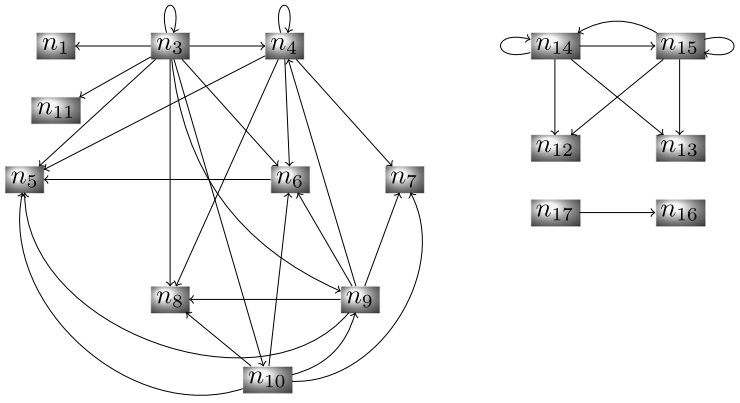


Fig. 2. Control dependency graph obtained from the CFG in Fig. 1 computed using postdominator relations.

may transform a terminating CFG into a nonterminating one. Thus, this kind of transformation is usual in practice. Moreover, we may obtain nonterminating CFG from nonterminating reactive systems, software models, or nonterminating distributed systems.

Let $N' = \{n_{18}, n_5, n_8\}$. Nodes n_4 and n_6 are weakly deciding vertices due to the N'-paths (i) $[n_4, n_8]$ and $[n_4 \dots n_5]$ and (ii) $[n_6, n_5]$ and $[n_6 \dots n_8]$. Thus, these nodes are not strongly committing according to Definition 9. There exist complete paths $[n_3, n_2]$ and $[n_{15}, n_{14}]$ which do not go through N'. Thus, n_3 and n_{15} are not strongly committing (Definition 9). Moreover, these nodes are not N'-avoiding. Thus, the strong control-closed superset of N' must be a superset of $X = \{n_3, n_4, n_6, n_{15}\}$. However, $N' \cup X$ is not yet a strong control closure of N' since now (i) n_9 and n_{10} are not $N' \cup X$-weakly deciding and consequently not strongly commiting, and (ii) they are not $N' \cup X$-avoiding. Nodes n_9 and n_{10} are not $N' \cup X$-weakly deciding due to the paths (i) $[n_9 \dots n_6]$ and $[n_9, n_3]$ and (ii) $[n_{10}, n_3]$ and $[n_{10} \dots n_6]$. Since all nodes in $X \cup \{n_9, n_{10}\}$ are reachable from $n_{18} \in N'$, $N' \cup \{n_3, n_4, n_6, n_9, n_{10}, n_{15}\}$ is the strong control closure of N'.

3. Motivations for efficient algorithms of computing WCC and SCC

Program slicing is one of the client applications of WCC and SCC. Slicing is specified by means of a *slicing criterion* which is usually a set of CFG nodes representing program points of interest. Static backward/forward program slicing then asks to select all program instructions that directly or indirectly affect/affected by the computation specified in the slicing criterion. Static dependence-based program slicing (Weiser, 1981; Lisper

et al., 2015; Khanfar et al., 2015) is performed by constructing a so-called program dependence graph (PDG) (Ferrante et al., 1987). A PDG explicitly represents the data and the control dependence relations in the control flow graph (CFG) of the input program. Any edge $n_1 \rightarrow n_2$ in a PDG represents either the control dependence relation $n_1 \overset{cd}{\rightarrow} n_2$ or the data dependence relation $n_1 \overset{dd}{\rightarrow} n_2$. The relation $n_1 \overset{dd}{\rightarrow} n_2$ holds if n_2 is using the value of a program variable defined at n_1. A PDG is constructed by computing all the data and the control dependence relations in the CFG of a program beforehand, and then include all edges (n, m) in the PDG if $n_1 \overset{dd}{\rightarrow} n_2$ or $n_1 \overset{cd}{\rightarrow} n_2$ holds. A forward/backward slice includes the set of all reachable nodes in the PDG from the nodes in the slicing criterion in the forward/backward direction.

The existence of the numerous kinds of control dependence in the literature puts us in the dilemma of which control dependence algorithm is to be used to construct PDG. Control dependence computation algorithms such as postdominator-based algorithms exist that cannot compute control dependencies from the following code having no exit point:

if *(p) { L1: x=x+1;* ***goto*** *L2; }* ***else*** *{ L2:* ***print****(x);****goto*** *L1; }*

Thus, for the above listing of code, PDG cannot be constructed if we use postdominator-based algorithms to compute control dependency, and consequently, we will not be able to apply PDG-based program slicing.

Algorithm 1 (*Slicing*)**.** Let C be the the slicing criterion, and let $S = C$.

1. $S' := \bigcup_{n \in S} \{m : m \overset{dd}{\rightarrow}{}^{*} n\}$
2. $S := cl(S')$
3. ***if*** $(S = S')$ ***then EXIT***
4. ***else GOTO*** *step 1*

Building a PDG by using a particular control dependence computation algorithm may miss computing certain kinds of control dependencies, and the program slicing may produce unsound results. With the advent of WCC and SCC, we obtain a more generalized method to compute control closure of a wide-range of programs. However, the above approach of static program slicing is not feasible with WCC and SCC. This is due to the fact that even though WCC and SCC capture/compute the weak and strong form of control dependencies that are nontermination (in)sensitive, it is not possible to tell specifically which node is control dependent on which other nodes. Given any set N' of CFG nodes, the weak/strong control closure $cl(N')$ of N' captures all control dependencies $n_1 \overset{cd}{\rightarrow} n_2$ such that $n_2 \in cl(N')$ implies $n_1, n_2 \in cl(N')$. However, by looking into the set $cl(N')$, it is not possible to tell if the relation $n_1 \overset{cd}{\rightarrow} n_2$ holds or not for any $n_1, n_2 \in cl(N')$. Since we cannot compute all individual control dependencies $n_1 \overset{cd}{\rightarrow} n_2$ beforehand, it is not possible to compute a PDG from a CFG using weak or strong control closed sets. However, Algorithm Algorithm 1 can be applied to perform the static program slicing using weak or strong control closures. The relation $\overset{dd}{\rightarrow}{}^{*}$ denotes the transitive-reflexive closure of $\overset{dd}{\rightarrow}$. The above algorithm computes the *slice set* S for backward slicing containing all CFG nodes that affect the computation at the nodes in C. For forward slicing, the relation $\overset{dd}{\rightarrow}{}^{*}$ has to be computed in the forward direction. To compute the relation $\overset{dd}{\rightarrow}{}^{*}$, we can build a data dependency graph (DDG) capturing only the data dependency relations. Then, step 1 in Algorithm 1 can be accomplished by obtaining the set of all reachable nodes in the DDG from the nodes in S in the forward/backward direction.

Algorithm 1 illustrates that step 2 needs to be performed iteratively until a fixpoint $S = S'$ is reached. Given any CFG (N, E), Danicic et al. provided expensive algorithms to compute weak and strong control closures with worst-case time complexity $O(|N|^3)$ and $O(|N|^4)$ respectively. These algorithms are not only computationally expensive, they cause the static forward/backward program slicing practically inefficient. In the following, we highlight the motivating factors of having WCC and SCC algorithms and the significance of their efficiency:

1. Traditional post-dominator based algorithms cannot capture control dependency relations from programs containing modern program instructions such as exceptions or nonterminating loops. WCC and SCC are the most generalized concepts to alleviate the limitations of post-dominator based control dependency computation from these programs.
2. Program slicing is one of the major clients of control dependency. However, the most widely used PDG-based program slicing cannot directly use WCC and SCC to capture control dependencies. Slicing algorithms like Algorithm 1 require repeated computation of WCC or SCC.
3. Efficient algorithms to compute WCC and SCC are desirable for their client applications. In particular, efficient WCC and SCC algorithms will improve the efficiency of program slicing and the client applications of program slicing such as debugging, software maintenance, program optimization, or program analysis.

In the next sections (Sections 4 and 5), we shall provide alternative simple yet practically efficient methods of computing minimal weak and strong control closed sets.

4. Efficient computation of minimal WCC

The relationship between WCC and weakly deciding vertices is the following (Lemma 51 in Danicic et al., 2011): the set of CFG nodes $N' \subseteq N$ is weakly control-closed in the CFG $G = (N, E)$ iff all N'-weakly deciding vertices in G that are reachable from N' are in N'. Moreover, $N' \cup WD_G(N')$ is the unique minimal weakly control-closed subset of N that contains N' (Theorem 54 in Danicic et al., 2011). We perform a simple and efficient two-step process of computing all N'-weakly deciding vertices $WD_G(N')$ followed by checking the reachability of these vertices from N' to compute the weakly control-closed subset of N containing N'.

In what follows, let $G = (N, E)$ be a CFG, let $N' \subseteq N$, and let $\mathcal{N}$ be the set of nodes such that $WD_G(N') \cup N' \subseteq \mathcal{N} \subseteq N$. The set of all N'-weakly deciding vertices $WD_G(N')$ are computed in the following two steps:

1. We compute a set of CFG nodes WD which is an overapproximation of the set of all N'-weakly deciding vertices, i.e., $WD_G(N') \subseteq WD$. The WD set includes all CFG nodes n such that n has two disjoint N'-paths. However, WD also contains spurious nodes having overlapping N'-paths or a single N' path which are not N'-weakly deciding. Thus, $\mathcal{N} = WD \cup N'$ is a weakly control-closed subset of N containing N' which is not minimal.
2. For each node $n \in WD$, the above process also indicates all CFG nodes $m \in \mathcal{N}$ such that either $[n \ldots m]$ is an N'-path or there exists an N'-path from n that must go through m. From this information, we build a directed graph $(\mathcal{N}, \mathcal{E})$ such that any edge $(n, m) \in \mathcal{E}$ indicates that n is possibly

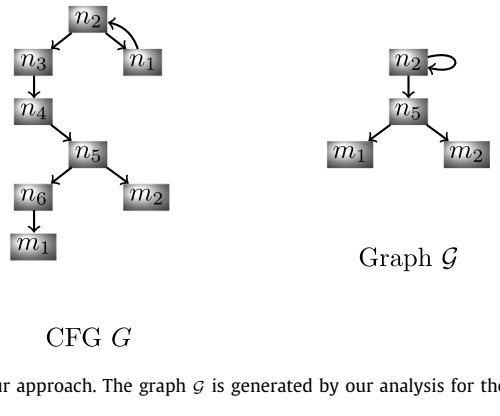


Fig. 3. CFG G used for the informal illustration of our approach. The graph $\mathcal{G}$ is generated by our analysis for the verification of potential N'-weakly deciding vertices.

Table 1
The $\mathcal{N}$-paths discovered by our algorithm. CFG nodes in Fig. 3 are visited in two different orders denoted by S_1 and S_2. T_i represents the sequence of visited CFG nodes and P_i represents the sequence of discovered $\mathcal{N}$-paths during the corresponding visits for $1 \leq i \leq 4$. The superscript on a path denotes its length.

S_1	T_1	=	$m_1 \rightarrow n_6 \rightarrow n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_2$
	P_1	=	$[m_1]^0$ $[n_6, m_1]^1$ $[n_5 \ldots m_1]^2$ $[n_4 \ldots m_1]^3$ $[n_3 \ldots m_1]^4$ $[n_2 \ldots m_1]^5$ $[n_1 \ldots m_1]^6$
	T_2	=	$m_2 \rightarrow n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_2$
	P_2	=	$[m_2]^0$ $[\mathbf{n_5}]^{\mathbf{0}}$ $[n_4, n_5]^1$ $[n_3 \ldots n_5]^2$ $[\mathbf{n_2}]^{\mathbf{0}}$ $[n_1 \ldots n_2]^1$
S_2	T_3	=	$m_1 \rightarrow n_6 \rightarrow n_5 \rightarrow n_4$
	P_3	=	$[m_1]^0$ $[n_6, m_1]^1$ $[n_5 \ldots m_1]^2$ $[n_4 \ldots m_1]^3$
	T_4	=	$m_2 \rightarrow n_5 \rightarrow n_4 \rightarrow n_3 \rightarrow n_2 \rightarrow n_1 \rightarrow n_2$
	P_4	=	$[m_2]^0$ $[\mathbf{n_5}]^{\mathbf{0}}$ $[n_4, n_5]^1$ $[n_3 \ldots n_5]^2$ $[n_2 \ldots n_5]^3$ $[n_1 \ldots n_5]^4$

a weakly deciding vertex, $m \in \mathcal{N}$, and there exists an $\mathcal{N}$-path $[n \ldots m]$ in G. Next, we perform a verification process to check that each node in WD has two disjoint N'-paths using the graph $(\mathcal{N}, \mathcal{E})$ and discard all nodes in WD that do not have two such paths. (See Fig. 3.)

4.1. An informal account of our approach

In this section, we give an informal description of our algorithm to compute the N'-weakly deciding vertices. The first step of this algorithm keeps track of all N'-paths (or $\mathcal{N}$-paths to be more specific where $\mathcal{N} = N'$ initially) in the CFG. We traverse the CFG backward from the nodes in N' and record all $\mathcal{N}$-paths at each visited node of the CFG. During this process, we discover all CFG nodes n that have more than one $\mathcal{N}$-paths ending at different CFG nodes, and n is included in WD (and thus $n \in \mathcal{N}$) as it is a potential N'-weakly deciding vertex. In the following, we illustrate this process using the CFG G in Fig. 3 where $m_1, m_2 \in N'$.

We have trivial $\mathcal{N}$-paths $[m_1]$ and $[m_2]$ of lengths zero at CFG nodes m_1 and m_2 respectively. The $\mathcal{N}$-paths from a node are identified from the $\mathcal{N}$-paths of its successor nodes. The trivial $\mathcal{N}$-path $[m_1]$ leads to the $\mathcal{N}$-path $[n_6, m_1]$ of length 1 which in turn leads to $[n_5 \ldots m_1]$ of length 2. Similarly, $[m_2]$ leads to the $\mathcal{N}$-path $[n_5, m_2]$ of length 1. Since two $\mathcal{N}$-paths $[n_5 \ldots m_1]$ and $[n_5, m_2]$ are identified from n_5, n_5 is included in WD and a new trivial $\mathcal{N}$-path $[n_5]$ of length 0 is identified. Different orders of visiting CFG nodes may produce different $\mathcal{N}$-paths.

Table 1 presents two possible orders of visiting the CFG nodes. The sequence of $\mathcal{N}$-paths denoted by P_1 is produced due to visiting the node sequence T_1. Note that an earlier visit to n_2 has produced the $\mathcal{N}$-path $[n_2 \ldots m_1]$ of length 5, and the last visit to n_2 from n_1 (via the backward edge) in T_1 does not produce any new $\mathcal{N}$-path at n_2 as it could generate the $\mathcal{N}$-path $[n_2 \ldots m_1]$ of length 7 which is not preferred over $[n_2 \ldots m_1]$ of length 5 by our algorithm. While visiting the sequence of nodes in T_2, our algorithm identifies two $\mathcal{N}$-paths $[n_5 \ldots m_1]$ and $[n_5 \ldots m_2]$, and thus it includes n_5 in WD. Moreover, a new trivial $\mathcal{N}$-path $[n_5]$ is generated, and the successive visits to the remaining sequence of nodes replace the old $\mathcal{N}$-paths by the newly generated paths of smaller lengths. From the $\mathcal{N}$-paths $[n_3 \ldots n_5]$ and $[n_1 \ldots m_1]$ at the successor nodes of n_2, our algorithm infers that there exist two $\mathcal{N}$-paths $[n_2 \ldots m_1]$ and $[n_2 \ldots n_5]$ from n_2, and thus it includes n_2 in WD even though no two disjoint $\mathcal{N}$-paths exist in G. Thus, WD is an overapproximation of $WD_G(N')$. When CFG nodes are visited according to the order specified in S_2, our algorithm does not infer two $\mathcal{N}$-paths at n_2, and thus it becomes more precise by not including n_2 in WD. Note that this order of visiting CFG nodes does not affect the soundness (as we prove it later in this section), but the precision and performance of the first step of our analysis, which is a well-known phenomenon in static program analysis. Note that our algorithm does not compute path lengths explicitly in generating $\mathcal{N}$-paths; it is accounted implicitly by our analysis.

The second step of our algorithm generates a graph $\mathcal{G}$ consisting of the set of nodes $N' \cup WD$ and the edges (n, m) such that $n \in WD$, $n' \in succ(n)$, and $[n' \ldots m]$ is the $\mathcal{N}$-path discovered in the first step of the analysis. Thus, $[n \ldots m]$ is an $\mathcal{N}$-path in the CFG. The graph $\mathcal{G}$ in Fig. 3 is generated from the WD set and the $\mathcal{N}$-paths generated due to visiting node sequences T_1 and T_2 in Table 1. Next, we traverse the graph $\mathcal{G}$ from N' backward; if a node $n \in WD$ is reached, we immediately know one of the $\mathcal{N}$-paths from n and explore the other unvisited branches of n to look for a second disjoint $\mathcal{N}$-path. For the graph $\mathcal{G}$ in Fig. 3, if $n_5 \in WD$ is reached from m_1, it ensures that $[n_5 \ldots m_1]$ is an $\mathcal{N}$-path in the CFG. Next, we look for a second $\mathcal{N}$-path in the other branch of n_5. In this particular case, the immediate successor of n_5 that is not yet visited is $m_2 \in N'$ such that $[n_5, m_2]$ is the second $\mathcal{N}$-path disjoint from $[n_5 \ldots m_1]$, which verifies that n_5 is an N'-weakly deciding vertex. We could have that $m_2 \notin N'$, and in that case, we traverse the graph $\mathcal{G}$ from m_2 in the forward direction to look for an $\mathcal{N}$-path different from $[n_5 \ldots m_1]$, include n_5 in $WD_G(N')$ if such a path is found, and excluded it from $WD_G(N')$ otherwise. Similarly, we discover the $\mathcal{N}$-path $[n_2 \ldots n_5]$ by reaching n_2 from n_5. However, since any $\mathcal{N}$-path from n_2 through the other branch of n_2 overlaps with $[n_2 \ldots n_5]$, n_2 is discarded to be a N'-weakly deciding vertex. When all nodes in WD are verified, we obtain the set $WD_G(N') \subseteq WD$ and the algorithm terminates.

4.2. An overapproximation of the weakly deciding vertices

We perform a backward traversal of the CFG from the nodes in N'. Initially, $\mathcal{N} = N'$. We maintain a function $A(n)$ for each CFG node $n \in N$. This function serves the following purposes:

1. If the backward traversal of the CFG visits only one $\mathcal{N}$-path $[n \dots m]$, then we set $A(n) = m$.
2. If two disjoint $\mathcal{N}$-paths $[n \dots m_1]$ and $[n \dots m_2]$ are visited during the backward traversal of the CFG, then we set $A(n) = n$.

We initialize the function $A(n)$ as follows:

$$A(n) \quad = \quad \begin{cases} \bot & n \in N \setminus N' \\ n & n \in N' \end{cases} \tag{1}$$

The valuation $A(n) = \bot$ indicates that no $\mathcal{N}$-path from n is visited yet. If we visit a CFG node $n \in N \setminus N'$ with two N'-paths (which may possibly be not disjoint due to overapproximation), then n is a potential N'-weakly deciding vertex. In this case, we set $A(n) = n$, n is included in WD (and hence $n \in \mathcal{N}$), and the function $A(n)$ will not be changed further.

If $A(n) \neq n$, then $A(n)$ may be modified multiple times during the walk of the CFG. If $A(n) = m_1$ is modified to $A(n) = m_2$ such that $n \neq m_1 \neq m_2$, then there exists a path $n, \dots, m_2, \dots, m_1$ in G such that $m_1, m_2 \in \mathcal{N}$ and $[n \dots m_2]$ is an $\mathcal{N}$-path in G. This may happen when (i) visiting the CFG discovers the $\mathcal{N}$-path $[n \dots m_1]$ such that $m_2 \notin \mathcal{N}$, and (ii) in a later visit to m_2, m_2 is included in WD (and in $\mathcal{N}$) that invalidates the path $[n \dots m_1]$ as an $\mathcal{N}$-path and obtains a new $\mathcal{N}$-path $[n \dots m_2]$. Note that if $[n \dots m]$ is an $\mathcal{N}$-path and $m \notin N'$, then there exists an N'-path from n that goes through m which we prove later in this section.

Algorithm 2 computes the set WD which is an overapproximation of weakly control-closed subset of N containing N'. It uses a worklist W to keep track of which CFG nodes to visit next. Note the following observations for Algorithm 2.

- For any node n in W, $A(n) \neq \bot$ due to the initializations in Eq. (1) and the update of A at steps 2(a) and 2(b) in Algorithm 2.
- The set S_m computed in line 6 is never empty due to the fact that n is a successor of m and $A(n) \neq \bot$.
- If $A(m) = m$, then m will never be included in W in steps 2(a) and 2(b) as further processing of node m will not give us any new information.
- Since m can only be included in WD in step (2a) if $A(m) \neq m$, and $A(m) = m$ for any $m \in N'$ due to Eq. (1), we must have $WD \cap N' = \emptyset$.
- Node m can only be included in W in step 2(b) if $A(m) = x$ is updated to $A(m) = y$ such that $y \neq x$.
- If any path $[n \dots m]$ is traversed such that $A(m) = m$ and no node in $[n \dots m] - \{m\}$ is in WD, then m is transferred such that $A(n') = m$ for all $n' \in [n \dots m] - \{m\}$ due to step (2b). Also, note that if $A(n) = m$, then we must have $A(m) = m$.
- The functions A are both the input and the output of the algorithm. This facilitates computing WD incrementally. This incremental WD computation will improve the performance of client applications of WCC such as program slicing (see Algorithm 1). The impact of incremental WD computation is further explained in the incremental computation of SCC in Section 5.2.

Theorems 1 and 2 below state the correctness of Algorithm 2 which we prove using an auxiliary lemma.

Lemma 1. *If $A(n) = m$ and $n \neq m$, then there exists an N'-path from n and all N'-paths from n must include m.*

Algorithm 2: OverapproxWD

```
Input  : G = (N, E), N', A
Output: A, WD
/* Step 1: Initialization */
1  WD = ∅
2  W = N'
   /* Step 2: Computation */
3  while (W ≠ ∅) do
4      Remove an element n from W
5      forall (m ∈ pred(n)) do
6          S_m = {A(m') : m' ∈ succ(m), A(m') ≠ ⊥}
           /* Step 2(a): Potential weakly deciding
              vertex */
7          if (|S_m| > 1 ∧ A(m) ≠ m) then
8              W = W ∪ {m}
9              A(m) = m
10             WD = WD ∪ {m}
           /* Step 2(b): Updating N'-paths */
11         if (|S_m| == 1 ∧ A(m) ≠ m ∧ x ∈ S_m) then
12             y = A(m)
13             A(m) = x
14             if (y ≠ x) then W = W ∪ {m}
15
```

Proof. Since $A(n) = m$, there exists a path $\pi = [n \dots m]$ visited in Algorithm 2 from m backward. The transfer of m to $A(n)$ is only possible if we have $S_x = \{m\}$ for all $x \in \pi - \{m\}$ and $A(x) = m$ is set in step (2b). Since $A(x) \neq x$, no node $x \in \pi - \{m\}$ is in $WD \cup N'$. Also, there exists a predecessor y of m such that $S_y = \{m\}$ which is only possible if $A(m) = m$. Thus, we must have $m \in N' \cup WD$ and π is a $(WD \cup N')$-path.

If $m \in N'$, then the lemma trivially holds. Suppose $m = m_1 \notin N'$. Then, we must have $m_1 \in WD$, and there exists a successor n_1 of m_1 such that $A(n_1) = m_2$. If $m_2 \notin N'$, then $m_2 \in WD$ and there exists a successor n_2 of m_2 such that $A(n_2) = m_3$. Thus, we obtain a subsequence of nodes $n_1, \dots, n_k$ such that $A(n_i) = m_{i+1}$ for all $1 \leq i \leq k$ and eventually we have $m_{k+1} \in N'$ since the CFG is finite and it is traversed from the nodes in N' backward. Thus, $[n \dots m_{k+1}]$ is an N'-path which go through m. □

Corollary 1. *If $A(n) = m$, then $m \in N' \cup WD$.*

Proof. The proof follows from the first part of the proof of Lemma 1. □

Theorem 1. *For any WD computed in Algorithm 2, $WD_G(N') \subseteq WD$.*

Proof. Suppose the lemma does not hold. So, there exists an N'-weakly deciding vertex $n \in WD_G(N')$ such that $n \notin WD$. Thus, there are two disjoint N'-paths from n. Let $n_1 = n, \dots, n_k$ and $m_1 = n, \dots, m_l$ be two N'-paths. Since $n_k, m_l \in N'$, $A(n_k) = n_k$ and $A(m_l) = m_l$ due to Eq. (1). Algorithm 2 traverses these paths and update $A(n_i)$ and $A(m_j)$ in step (2b) such that

$A(n_i) \neq \bot$ and $A(m_j) \neq \bot$ for all $1 \leq i < k$ and $1 \leq j < l$.

Since $n \notin WD$, $|S_n| \leq 1$ in line 6 in Algorithm 2. Node n has at most two successors according to the definition of CFG (Definition 1). Since $A(n_2) \neq \bot$ and $A(m_2) \neq \bot$, $|S_n| \leq 1$ is only possible if $A(n_2) = A(m_2)$. Let $A(n_2) = m$. Then, we must have $A(n) = m$ and all N'-paths must include m according to Lemma 1. Thus, we conclude that n is not an N'-weakly deciding vertex since the N'-paths from n are not disjoint, and we obtain the contradiction. □

Theorem 2. *Algorithm 2 eventually terminates.*

Proof. Algorithm 2 iterates as long as there exist elements in W. For all $n \in N$ such that $A(n) = n$, n is included in WD and it never gets included in W again. If the value of $A(n)$ remains $\perp$, then n is never reached and included in W during the walk of the CFG. Thus the algorithm can only be nonterminating for some node n such that $A(n) \neq n \neq \perp$. According to step (2b) in the algorithm, n can only be included in W if the new value of $A(n)$ is different from the old one. Thus, in order for the algorithm to be nonterminating, there exists an infinite update to $A(n)$ by the sequence of values $m_1, \ldots, m_k, \ldots$ such that no two consecutive values are the same, i.e., $m_i \neq m_{i+1}$ for all $i \geq 1$.

According to Lemma 1, $A(n) = m_i$ implies that there exists an N'-path from n and all N'-paths from n must include m_i. Thus, there exists a path $[n \ldots m_i]$ in the CFG. If $A(n)$ is updated by m_{i+1}, then $m_{i+1} \in WD$ and $A(m_{i+1}) = m_{i+1}$. Node m_{i+1} must be in the path $[n \ldots m_i]$ as otherwise we eventually have $S_n = \{m_i, m_{i+1}\}$ in line 6 which will lead to $A(n) = n$. So, $A(n)$ will never become m_i again in (2b) as all N'-paths from n must go through $m_{i+1} \in WD$. Similarly, if $A(n)$ is updated by m_{i+2}, m_{i+2} must be in the path $[n \ldots m_{i+1}]$ and $A(n)$ will never be updated by m_{i+1} again. Since the path $[n \ldots m_{i+1}]$ has a finite number of nodes, $A(n)$ cannot be updated infinitely, and the algorithm eventually terminates. □

Example 4. Let $N' = \{n_5, n_8\}$ for the CFG in Fig. 1. Algorithm 2 computes A and WD as follows:

- $A(n) = \perp$ for $n \in \{n_0, n_2\}$
- $A(n_i) = n_i$ for $i \in \{4, \ldots, 6, 8, \ldots, 10, 14, 15, 17\}$
- $A(n_i) = n_{10}$ for $i \in \{1, 3, 11, 13\}$
- $A(n_7) = n_6, A(n_{12}) = n_{15}, A(n_{16}) = n_{15}, A(n_{18}) = n_{17}$
- $WD = \{n_4, n_6, n_9, n_{10}, n_{14}, n_{15}, n_{17}\}$

Note that CFG nodes $n_9, n_{10}, n_{14}, n_{15}$, and n_{17} have no disjoint N'-paths as all N'-paths from these nodes must go through n_{10}. Thus, these nodes do not belong to $WD_G(N')$. However, we have $WD_G(N') = \{n_4, n_6\}$ and $WD_G(N') \subseteq WD$ holds.

4.3. Generating minimal weakly deciding vertices

Algorithm 2 is sound according to Theorem 1. However, as illustrated in Section 4.1, the WD set computed in this algorithm may contain spurious nodes that are not N'-weakly deciding. In what follows, we provide a general and efficient algorithm to verify the results obtained from Algorithm 2 and discard all incorrectly identified N'-weakly deciding vertices. Thus, both algorithms together provide minimal and sound N'-weakly deciding vertices. We first represent the solutions generated by Algorithm 2 as a directed graph $\mathcal{G}$ as follows:

Definition 12. $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is a directed graph, where

- $\mathcal{N} = N' \cup WD$, and
- $\mathcal{E} = \{(n, A(m)) : n \in WD, m \in succ(n), A(m) \neq \perp\}$.

Note that $succ(n)$ is the set of successors of n in the CFG. In Fig. 4, $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is constructed from A and WD in Example 4 and the CFG in Fig. 1. Any graph $\mathcal{G}$ constructed according to Definition 12 has the following properties:

- If there exists an edge (n, m) in $\mathcal{E}$ such that $m \in N'$, then there exists an N'-path $[n \ldots m]$ in the CFG G.
- An edge (n, m) in $\mathcal{E}$ such that $m \in WD$ implies that there exists an N'-path from n going through m (from Lemma 1).
- There exist no successors of a node in N' since $WD \cap N' = \emptyset$.
- Graph $\mathcal{G}$ may be an edge-disjoint graph since there may exist N'-weakly deciding vertices and their N'-paths do not overlap.

Algorithm 3: VerifyWDV

Input : $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and N'
Output: WD_{min}
/* Initialization */
1 **forall** $(n \in \mathcal{N} \setminus N')$ **do**
2 $\quad \bar{A}(n) = \perp$, $V(n) = false$, and $T(n) = false$
3 **forall** $(n \in N')$ **do**
4 $\quad \bar{A}(n) = n$, $V(n) = true$, and $T(n) = true$
5 $W = \bigcup_{n \in N'} pred_{\mathcal{G}}(n)$
6 $WD_{min} = \emptyset$
/* Computation */
7 **while** $(W \neq \emptyset)$ **do**
8 $\quad$ Remove n from W and set $V(n) = true$
9 $\quad S_n = \{\bar{A}(m) : m \in succ_{\mathcal{G}}(n), \bar{A}(m) \neq \perp\}$
10 $\quad R_n = \{m : m \in succ_{\mathcal{G}}(n), \bar{A}(m) = \perp\}$
11 $\quad$ Let $m \in S_n$
12 $\quad$ **if** $((|S_n| > 1) \vee disjointNPath(m, \mathcal{G}, R_n, WD_{min}, N'))$ **then**
13 $\quad\quad \bar{A}(n) = n$
14 $\quad\quad$ **if** $(T(n) = false)$ **then** $WD_{min} = WD_{min} \cup \{n\}$
15 $\quad$ **else** $\bar{A}(n) = \bar{A}(m)$
16 $\quad$ **forall** $(n' \in pred_{\mathcal{G}}(n)$ *such that* $V(n') = false)$ **do**
17 $\quad\quad W = W \cup \{n'\}$

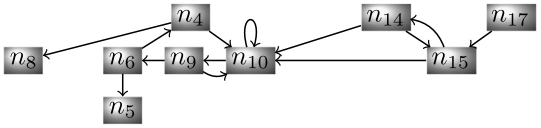


Fig. 4. Graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ constructed according to Definition 12 from the CFG in Fig. 1, and A and WD in Example 4.

- Since our CFG has at most two successors according to Definition 1, any node in $\mathcal{G}$ has at most two successors. However, some nodes in $\mathcal{G}$ may have self-loop or only one successor due to the spurious nodes generated in WD. Moreover, $|\mathcal{N}| \leq |N|$, $|\mathcal{E}| \leq |E|$.

The intuitive idea of the verification process is the following. For any $n \in N'$, we consider a predecessor m of n in $\mathcal{G}$. Thus, we know that $[m \ldots n]$ is an N'-path in the CFG G. If there exist another successor $n' \in N'$ of m such that $n \neq n'$, then $[m \ldots n']$ is another N'-path disjoint from $[m \ldots n]$ and m is an N'-weakly deciding vertex. However, all other successors of m might be from WD instead of N'. Let $succ_{\mathcal{G}}(m)$ and $pred_{\mathcal{G}}(m)$ be the sets of successors and predecessors of m in $\mathcal{G}$. Then, we traverse $\mathcal{G}$ from the nodes in $succ_{\mathcal{G}}(m) \setminus N'$ in the forward direction to find an N'-path from m which is disjoint from $[m \ldots n]$. If it visits a node in N' different from n, then m is an N'-weakly deciding vertex due to having two disjoint N'-paths. Otherwise, we exclude m from WD. Most nodes in WD can be immediately verified by looking into their immediate successors without traversing the whole graph $\mathcal{G}$. Also, the graph $\mathcal{G}$ is usually much smaller than the CFG. Thus, the whole verification process is practically very efficient.

Given the graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, Algorithm 3 generates WD_{min} which is the set of minimal N'-weakly deciding vertices. Like Algorithm 2, we use a function $\bar{A}(n)$ to keep track of N'-paths visited from n. Initially, $\bar{A}(n) = \perp$ for all $n \in \mathcal{N} \setminus N'$ and $\bar{A}(n) = n$ otherwise. A boolean function $T(n)$ is set to true if $n \in N'$, and $T(n) = false$ otherwise. Another boolean function $V(n)$, which is initially *false*, is set to *true* if n is already verified. The procedure $disjointNPath(m, \mathcal{G}, R_n, WD_{min}, N')$ used in the algorithm traverses

Algorithm 4: computeWD

Input : $G = (N, E), N'$
Output: WD
1 Apply Eq. (1) to initialize A
2 $(A, WD) = OverapproxWD(G, N', A)$
3 Construct $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ according to Definition 12
4 $WD = VerifyWDV(\mathcal{G}, N')$

Algorithm 5: minimalWCC

Input : $G = (N, E), N'$
Output: WCC
1 $WD = computeWD(\mathcal{G}, N')$
2 WCC =(All reachable nodes in WD from N') $\cup N'$

the graph $\mathcal{G}$ from the nodes in R_n in the forward direction visiting each node at most once. If a node in $N' \cup WD_{min}$ different from m is visited, then it returns *true*, otherwise *false*. During this traversal, no successors of a node in $N' \cup WD_{min}$ are visited as an N'-path must end at a node in N'. We skip providing the details of this procedure since it is a simple graph traversal algorithm. Note that $S_n \neq \emptyset$ at Line 9. This is because there exists a successor m of n from which n is reached during the backward traversal of the graph $\mathcal{G}$ and $\bar{A}(m) \neq \perp$.

Theorem 3 below proves that WD_{min} is the minimal weakly control-closed subset of N containing N'.

Theorem 3. *For any WD_{min} computed in Algorithm 3, $WD_G(N') = WD_{min}$.*

Proof. "$\subseteq$": Let $n \in WD_G(N')$. According to Theorem 1, $WD_G(N') \subseteq WD$ and thus $n \in WD$. Suppose $m_1, m_2 \in succ(n)$ since there exist two disjoint N'-paths from n, and also assume that $A(m_i) = n_1^i$ for $i = 1, 2$. Thus, (n, n_1^i) is an edge in $\mathcal{G}$ for $i = 1, 2$. According to Corollary 1, $n_1^i \in N' \cup WD$. If $n_1^i \notin N'$, we can show similarly that there exists a node n_2^i such that (n_1^i, n_2^i) is an edge in $\mathcal{G}$ for some $1 \leq i \leq 2$ and $n_2^i \in N' \cup WD$. Since graph $\mathcal{G}$ and the CFG G are finite, eventually we have the following sequence of edges

$$(n, n_1^1), (n_1^1, n_2^1), \ldots, (n_{k-1}^1, n_k^1) \text{ and } (n, n_1^2), (n_1^2, n_2^2), \ldots, (n_{l-1}^2, n_l^2)$$

such that $n_k^1, n_l^2 \in N'$ for some $k, l \geq 1$. The graph $\mathcal{G}$ is traversed backward from $n_k^i \in N'$ and n will be reached in successive iterations in Algorithm 3. Thus, n is reached by traversing an N'-path $[n \ldots n_k^1]$ backward. Either another N'-path $[n \ldots n_l^2]$ will be discovered immediately during the construction of S_n at line (9) of the algorithm or it will be discovered by calling the procedure *disjointNPath* and we eventually have $n \in WD_{min}$.

"$\supseteq$": Let $n \in WD_{min}$. Thus, there exists a node $m \in N'$ such that n is reached during traversing the graph $\mathcal{G}$ backward and thus $[n \ldots m]$ is an N'-path. Also, there exists a successor $m' \neq m$ of n such that either $m' \in N'$ or *disjointNPath* procedure traverses an N'-path from m_2 which is disjoint from $[n \ldots m]$. Thus, $n \in WD_G(N')$ due to having two disjoint N'-paths. □

4.4. Computing minimal WCC

After obtaining the WD_{min} set containing minimal N'-weakly deciding vertices, computing minimal WCC requires checking the reachability of these nodes from the nodes in N'. Algorithms 4 and 5 below provide the complete picture of computing minimal WCC.

Example 5. For the graph $\mathcal{G}$ in Fig. 4 and $N' = \{n_5, n_8\}$, Algorithm 3 generates $WD_{min} = \{n_4, n_6\}$. Algorithm 5 computes $WCC = \{n_4, n_5, n_6, n_8\}$ for the CFG in Fig. 1 and N' as above.

4.5. Worst-case time complexity

Lemma 2. *The worst-case time complexity of Algorithm 2 is $O(|N|^2)$.*

Proof. The worst-case time complexity is dominated by the costs in step (2) of Algorithm 2. Since $|succ(n)| \leq 2$ for any CFG node n, all the operations in steps (2a)–(2b) have constant complexity. However, after removing a node n from W, all the predecessors of n are visited. If the CFG G has no N'-weakly deciding vertices, then Algorithm 2 visits at most $|N|$ nodes and $|E|$ edges after which the operation $y \neq x$ in (2b) is always false, no node will be inserted in W, and thus the cost will be $O(|N| + |E|)$. In order to obtain a vertex in WD, it needs to visit at most $|N|$ nodes and $|E|$ edges and the maximum cost will be $O(|N| + |E|)$. If a node n is included in WD, then we set $A(n) = n$ and n will never be included in W afterwards due to the first conditional instruction in step (2b). Since we can have at most $|N|$ N'-weakly deciding vertices, the total worst-case cost will be $O((|N|+|E|)*|N|)$. Since any CFG node has at most two successors, $O(|E|) = O(|N|)$, and thus the worst-case time complexity is $O(|N|^2)$. □

Lemma 3. *The worst-case time complexity of Algorithm 3 is $O(|N|^2)$.*

Proof. The initialization steps in Algorithm 3 have the worst-case cost $O(|\mathcal{N}|)$. The worst-case cost of Algorithm 3 is dominated by the ***while*** loop (line 7–17). This loop iterates at most $|\mathcal{N}|$ times since (i) once an element is removed from W, it is marked as visited and never inserted into W again, and (ii) the loop iterates as long as there are elements in W. Computing the sets S_n and R_n have constant costs since $|succ_{\mathcal{G}}(n)| \leq 2$. The costs of all other operations between lines 8–15 are also constant except the *disjointNPath* procedure which has the worst-case cost of $O(|\mathcal{N}| + |\mathcal{E}|)$ as it is a simple forward graph traversal algorithm visiting each node and edge at most once and other operations have constant cost. The ***forall*** loop at line 16–17 visits the edges in $\mathcal{E}$ to insert elements in W and cannot visit more than $|\mathcal{E}|$ edges. Thus, the dominating cost of Algorithm 3 is $O((|\mathcal{N}| + |\mathcal{E}|) \times |\mathcal{N}|)$. Since $|\mathcal{N}| \leq N$, $|\mathcal{E}| \leq E$, and $O(|N|) = O(|E|)$, $O(|N|^2)$ is the worst-case time complexity of this algorithm. □

Lemma 4. *The worst-case time complexity of Algorithm 4 is $O(|N|^2)$.*

Proof. The worst-case time complexity of Algorithm 4 is dominated by the *VerifyWDV* and *OverapproxWD* procedures which have the worst-case time complexity $O(|N|^2)$ according to Lemmas 2 and 3. □

Theorem 4. *The worst-case time complexity of Algorithm 5 is $O(|N|^2)$.*

Proof. The worst-case time complexity of Algorithm 5 is dominated by the *computeWD* procedure as checking the reachability of a set of nodes in a graph $G = (N, E)$ can be performed in linear time (i.e. $O(|N|)$ in the worst case). Since $O(|N|^2)$ is the worst-case complexity of the *computeWD* procedure according to Theorem 4, the worst-case time complexity of computing minimal WCC in Algorithm 5 is $O(|N|^2)$. □

5. Efficient computation of minimal SCC

5.1. Theoretical foundation and algorithms to compute minimal SCC

Let $G = (N, E)$ be a CFG and let $N' \subseteq N$. According to Definition 11, N' is not strongly control closed if and only if there exists a node $n \in N \setminus N'$ such that n is reachable from N' which is neither N'-strongly committing nor N'-avoiding. Any

strongly control-closed superset of N' should include such n. In what follows, we identify all CFG nodes n to be included in the strong control closure of N'.

If any CFG node $n \in N \setminus N'$ is not N'-strongly committing, then according to Definition 6, either or both of the following conditions are true:

1. n is not N'-weakly committing,
2. not all complete paths from n contain an element of N'.

If n is not N'-avoiding, then according to Definition 10,

3. there exists a node in N' that is reachable from n.

The first condition implies that there exist two disjoint N'-paths from n according to Definition 6, and thus n is an N'-weakly deciding vertex. We use Algorithms 2 and 3 to identify all weakly deciding vertices. To identify all nodes satisfying conditions (2) and (3) above, we compute the set $\Gamma(G, N')$ defined according to Danicic et al. (2011) as follows:

Definition 13. Let $G = (N, E)$ be a CFG and let $N' \subseteq N$. We define $\Gamma(G, N')$ to be the set of all $n \in N$ that lie on a complete path in G which does not pass through N'.

The set $\Gamma(G, N')$ in Definition 13 contains all CFG nodes n from which there exists at least one complete path not containing an element from N'. Algorithm 6 computes $\Gamma(G, N')$ which has the linear worst-case execution time complexity as proved in Section 5.3. We denote $\Gamma(G, N')$ as Γ for simplicity when G and N' are understood from the context. Algorithm 6 is based on the simple idea of considering all CFG nodes to be in the set Γ at the beginning, and discard all CFG nodes n from Γ when the backward traversal of the CFG G identifies that all complete paths from n include a node from N'. This procedure is done as follows:

- We initialize $\gamma(n) = \mathit{true}$ for all CFG nodes $n \in N$ to indicate that n belongs to the set Γ.
- We traverse the CFG from the nodes in N' in the backward direction. A worklist W is maintained to keep track of all CFG nodes to be visited. W is initialized by N'.
- We set $\gamma(n) = \mathit{false}$ for all $n \in N'$ as all complete paths from n include a node from N', and thus $n \in N'$ does not belong to Γ;
- If $\gamma(m) = \mathit{false}$ for all successors $m \in \mathit{succ}(n)$ of any CFG node n, then all complete paths from n include a node from N'. Then, $\gamma(n)$ is set to false and G is traversed from n backward if $\gamma(n)$ is not set to false before.
- If there exists a successor m of n such that $\gamma(m) = \mathit{true}$, then there possibly exists a complete path from n not including a node from N', and thus no changes to $\gamma(n)$ are performed.

Algorithm 6 visits each CFG node n exactly once possibly modifying $\gamma(n)$, and Γ is finally constructed from γ.

We provide the following lemma stating a set of conditions to be tested for detecting all CFG nodes n that are not N'-strongly committing and not N'-avoiding, and to be included in the strong control closure of N'.

Lemma 5. *Let $G = (N, E)$ be a CFG and let $N' \subseteq N$. N' is not strongly control-closed if there exists a CFG node n such that the following conditions hold:*

1. $n \in N \setminus N'$,
2. *n is reachable in G from N',*
3. $n \in WD_G(N') \cup \Gamma$, *and*
4. *there exists an edge $(n, m) \in E$ such that $\gamma(m) = \mathit{false}$.*

Algorithm 6: compute Γ

```
Input  : G = (N, E), N'
Output: Γ, γ
1 forall (n ∈ N) do γ(n) = true
2 W = N'
3 while (W ≠ ∅) do
4     Remove n from W
5     γ(n) = false
6     forall (m ∈ pred(n)) do
7         if (∀n' ∈ succ(m).not(γ(n')) ∧ γ(m)) then
8             W = W ∪ {m}
9 Γ = {n : n ∈ N, γ(n)}
```

Proof. If N' is not strongly control-closed, then there exists a CFG node $n \in N \setminus N'$ that is reachable in G from N' and are neither N'-strongly committing nor N'-avoiding according to Definition 11. Thus, conditions (1) and (2) of the lemma are satisfied for n. If n is not N'-strongly committing, then either or both of the conditions are true: (1) n is not N'-weakly committing, (2) not all complete paths from n contain an element of N'. In the former case, n is a weakly deciding vertex according to Definitions 7 and 8, and thus $n \in WD_G(N')$. In the latter case, $n \in \Gamma$, and thus the third condition of the lemma is true for n. If n is not N' avoiding, then there exists a complete path from n that passes through a node in N'. Suppose $n_0 = n, n_1 = m, \ldots, n_k \in N'$ be an N'-path. We choose n such that $k \geq 1$ is minimal where $n_i \notin \Gamma$ for all $1 \leq i \leq k$. If any $n_i \in \Gamma$ exists, then we choose $n = n_i$ instead of $n = n_0$. Thus, there exist N'-paths from all successors of each n_i. This implies that $\gamma(n') = \mathit{false}$ for all successors n' of n_i, and consequently, $\gamma(n_i)$ is set to false according to Algorithm 6. Thus, condition (4) of the lemma is also satisfied. □

Lemma 6. *Let $G = (N, E)$ be a CFG and let $N' \subseteq N$. Every strong control closure of N' must include the CFG node n satisfying conditions (1)–(4) in Lemma 5.*

Proof. Let X be the strong control closure of N' and let there exist a CFG node $n \notin X$ satisfying conditions (1)–(4) in Lemma 5. Condition (3) in Lemma 5 implies the following contradictions:

- $n \in WD_G(N')$: There exist two disjoint N'-paths

$\pi_1 \equiv n_0 = n, n_1, \ldots, n_k \in N'$ and $\pi_2 \equiv m_0 = n, m_1, \ldots, m_l \in N'$

for $k, l \geq 1$ such that $n_i \neq m_j$ for all $1 \leq i \leq k$ and $1 \leq j \leq l$. Since $\{n_k, m_l\} \subseteq N' \subseteq X$, and $n \notin X$, there exist minimal indexes i and j such that $\{n_i, m_j\} \subseteq X$, and thus $n \in WD_G(X)$. So, n is not X-weakly committing. Moreover, it is not X-avoiding due to the paths π_1 or π_2 as $N' \subseteq X$. Thus, X is not a strong control closure of N'.

- $n \in \Gamma$: There exists a path $\pi_1 = n_0 = n, n_1, \ldots$ such that $n_i \notin N'$. Condition (4) in Lemma 5 implies that there exists an N'-paths $m_0 = n, m_1, \ldots, m_k \in N'$. Since $N' \subseteq X$, there exists a minimal index $1 \leq i \leq k$ such that $m_i \in X$ and $[m_0 \ldots m_i]$ is an X-path. If no n_j is in X, then n is not X-strongly committing, and hence X is not a strong control closure. If there exists a minimal index j such that $n_j \in X$, then $[n_0 \ldots n_j]$ and $[m_0 \ldots m_i]$ are two W-paths and hence $n \in WD_G(X)$. Thus, n is not X-strongly committing, and hence X is not a strong control closure. □

Lemma 7. *Let $G = (N, E)$ be a CFG and let $N' \subseteq N$. If N' is strongly control closed, then no CFG node n exists satisfying conditions (1)–(4) in Lemma 5.*

Algorithm 7: computeMinimalSCC

Input : $G = (N, E), N'$
Output: SCC

1 $SCC = N'$
2 **while** $(true)$ **do**
3 $\quad WD = computeWD(G, SCC)$
4 $\quad (\Gamma, \gamma) = compute\Gamma(G, SCC)$
5 $\quad X = \{n : n \in WD \cup \Gamma \setminus SCC, \exists m \in succ(n).not(\gamma(m))\}$
6 $\quad Y = \{n : n \in X, n \text{ is reachable from } SCC\}$
7 $\quad$ **if** $(Y == \emptyset)$ **then** ***break***
8 $\quad SCC = SCC \cup Y$

Table 2
Computing $\Gamma(G_1, N')$ according to Algorithm 6 where G_1 is specified in Example 3 and $N' = \{n_{18}, n_5, n_8\}$.

Iteration	W	n	$m \in pred(n)$ included in W	Action
1	$\{n_5, n_8, n_{18}\}$	n_{18}	–	$\gamma(n_{18}) = false$
2	$\{n_5, n_8\}$	n_5	–	$\gamma(n_5) = false$
3	$\{n_8\}$	n_8	n_4	$\gamma(n_8) = false$
4	$\{n_4\}$	n_4	n_6	$\gamma(n_4) = false$
5	$\{n_6\}$	n_6	n_7	$\gamma(n_6) = false$
6	$\{n_7\}$	n_7	–	$\gamma(n_7) = false$

Proof. Let N' be a strongly control closed set. Assume that there exists a node n in G satisfying conditions (1)–(4) in Lemma 5. Then, $WD_G(N') = \emptyset$ as otherwise N' is not weakly committing, and consequently, it is not strongly committing, and hence N' is not strongly control-closed. Since n satisfies condition (3) in Lemma 5, we must have $n \in \Gamma$. Then, there exists a complete path $n_0 = n, n_1, \ldots$ which does not go through a node in N'. Thus, n is not N'-strongly committing. As n satisfies the condition (4) in Lemma 5, there exists an edge $(n, m) \in E$ such that $\gamma(m) = false$. So, there exists an N'-path $m_0 = n, m_1 = m, \ldots, m_k \in N'$, and thus n is not N'-avoiding. Consequently, our assumption leads to the contradiction that N' is not strongly control closed. □

Algorithm 7 computes the strong control closure SCC of N' based on Lemma 5–7. We provide the following lemma stating that SCC is the unique minimal strong control closure.

Lemma 8. *Let $G = (N, E)$ be a CFG, let $N' \subseteq N$, and let SCC be the set computed in Algorithm 7. Then, SCC is the unique minimal strong control closure of N'.*

Proof. Let Y_i be the set of CFG nodes computed at the ith iteration of the ***while*** loop in Algorithm 7 for $i \geq 1$. We have $Y_i \cap Y_j = \emptyset$ for all $i \neq j$ since if any Y_i is included in SCC, no nodes in Y_i are ever considered in the subsequent iterations j to be included in Y_j. Since N is finite, eventually, there exists $k \geq 1$ such that $Y_k = \emptyset$, and the ***while*** loop terminates after the kth iteration. Let $SCC_0 = N'$, $SCC_i = SCC_0 \bigcup_{j=1}^{i} Y_j$, and $SCC = SCC_{k-1}$.

The unique minimality of SCC is obtained due to Lemmas 6 and 7 as follows. If SCC_i is not strongly control-closed, then each CFG node $n \in Y_{i+1}$ must be included in any strong control closure of N' for all $0 \leq i \leq k-2$. Eventually, we obtain the strong control closure $SCC = SCC_{k-1}$, and obtain Y_k as an empty set due to Lemma 7. □

Example 6. Consider the CFG G_1 in Example 3. It is the CFG in Fig. 1 with nodes n_0, n_{12}, and n_{13} and their associated edges removed. Let $N' = \{n_5, n_8, n_{18}\}$. Table 2 lists the results of an execution of the ***while*** loop in Algorithm 6 for the CFG G_1 and the set N'. The third column lists the CFG node n that is removed from W in each iteration, the fourth column lists all CFG node $m \in pred(n)$ that satisfies the ***if*** condition at line 7 in the algorithm and are included in the worklist W, and the final column specifies which γ function is set to *false* . Except the value of $\gamma(n)$ listed in the final column, $\gamma(n) = true$ for all other remaining CFG nodes from which we obtain $\Gamma = \{n_1, n_2, n_3, n_9, n_{10}, n_{10}, n_{14}, \ldots, n_{17}\}$.

Table 3
Steps of computing SCC according to Algorithm 7 for the CFG G_1 specified in Example 3 and the set $N' = \{n_{18}, n_5, n_8\}$.

Iteration	SCC	WD	$\Gamma(G_1, SCC)$	Y
1	N'	$\{n_4, n_6\}$	$\{n_1, n_2, n_3, n_9, n_{10}, n_{11}, n_{14}, \ldots, n_{17}\}$	$\{n_4, n_6, n_9\}$
2	$N' \cup Y_1$	$\emptyset$	$\{n_1, n_2, n_3, n_{10}, n_{11}, n_{14}, \ldots, n_{17}\}$	$\{n_{10}\}$
3	$N' \cup Y_1 \cup Y_2$	$\emptyset$	$\{n_2, n_3, n_{14}, \ldots, n_{17}\}$	$\{n_3, n_{15}\}$
4	$N' \cup Y_1 \cup \cdots \cup Y_3$	$\emptyset$	$\{n_2, n_{14}\}$	$\emptyset$

Table 3 lists the sets SCC, WD, Γ, and Y computed at different iterations of the ***while*** loop in Algorithm 7. The set Y computed at iteration i is denoted by Y_i in the table. We only have shown the steps of computing Γ at the first iteration in Table 2. We obtain Γ at other iterations by using similar steps. The WD set is computed according to Algorithm 4 which can easily be verified by looking into the SCC paths in the CFG G_1. The final column lists the set Y computed at each iteration which illustrates that no node is listed twice in this column, and eventually we obtain $Y_4 = \emptyset$ and the ***while*** loop terminates. Finally, we obtain $SCC = \{n_3, n_4, n_5, n_6, n_8, n_9, n_{10}, n_{15}, n_{18}\}$ which is the strong control closure of N'. In order to see the difference between WCC and SCC, the WCC of N' is $\{n_4, n_5, n_6, n_8, n_{18}\}$. Since the WCC does not consider nontermination, it is usually much smaller than SCC.

5.2. Incremental computation of SCC

The strongly control-closed set SCC in Algorithm 7 is computed iteratively. In each iteration of the ***while*** loop, SCC is augmented by a distinct set Y which is a subset of the weakly deciding vertices, and the set Γ, computed by calling the procedures *computeWD* and *compute*Γ. Let $Y_1, \ldots, Y_k$ be the distinct sets such that Y_i is computed at the ith iteration of the ***while*** loop. We have $Y_i \neq Y_j$ if $i \neq j$ since each Y_i is included in SCC and during the computation of the set Y_j at subsequent iterations, the elements of SCC are discarded in computing Y_j. Thus, SCC is progressively growing by distinct nodes in each iteration, and eventually, it is a strongly control-closed set.

However, this iterative procedure of computing SCC performs redundant computation by the *computeWD* (more specifically the *OverapproxWD* procedure called within the *computeWD* procedure in computing weakly deciding vertices) and the *compute*Γ procedures. The *OverapproxWD* procedure traverses the potential N'-paths by visiting the nodes in the CFG backward from the set of nodes N'. The SCC set is progressive growing from the initial set $SCC_0 = N'$ to $SCC_i = N' \bigcup_{j=1}^{i} Y_i$ at the ith iteration, and the *OverapproxWD* procedure is called to compute the weakly deciding vertices of the set SCC_{i-1} at the ith iteration of the ***while*** loop in Algorithm 7. Thus, some SCC_{i-1}-paths are visited more often than necessary, and visiting these paths do not give us more information. For example, if the ***while*** loop in Algorithm 7 terminates after k iteration, then N'-paths are visited at least k times, Y_1-paths are visited at least $k-1$ times, Y_2-paths are visited at least $k-2$ times, and so on.

All potential weakly deciding vertices can be computed more efficiently by considering the following facts. If $A(n) = m$ is computed by any call to the *OverapproxWD* procedure at any iteration j of the ***while*** loop in Algorithm 7, then there exists a path $\pi \equiv n_0 = n, \ldots, n_l = m \in SCC_{j-1}$ and $A(n_t) = m$ is

Algorithm 8: computeWDIncr

Input : $G = (N, E), N', A$
Output: A, WD

1 $(A, WD) = OverapproxWD(G, N', A)$
2 Construct $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ according to Definition 12
3 $WD = VerifyWDV(\mathcal{G}, N')$

Algorithm 9: computeΓIncr

Input : $G = (N, E), N', \gamma$
Output: Γ, γ

1 $W = N'$
2 **while** $(W \neq \emptyset)$ **do**
3 Remove n from W
4 $\gamma(n) = false$
5 **forall** $(m \in pred(n))$ **do**
6 **if** $(\forall n' \in succ(m).not(\gamma(n')) \wedge \gamma(m))$ **then**
7 $W = W \cup \{m\}$
8 $\Gamma = \{n : n \in N, \gamma(n)\}$

also computed for all $0 \leq t \leq l$ as $[n_t \dots n_l]$ is also a potential SCC_{j-1}-path. If no CFG node in the path π is ever included in Y_i in the subsequent iteration i of the ***while*** loop in Algorithm 7, then the fact $A(n_t) = m$ does not require any changes. However, if any node n_z for any $1 \leq z \leq l-1$ is included in Y_i, then the fact $A(n_t) = m$ will not be changed for all $z+1 \leq t \leq l$ as $[n_t \dots n_l]$ is still a potential SCC_{i-1}-path. But, we will obtain new facts $A(n_t) = n_z$ for all $0 \leq t \leq z$ as $[n_t \dots n_z]$ is a potential SCC_{i-1}-path. Thus, it is enough to call the *OverapproxWD* procedure with the argument Y_{i-1} instead of SCC_{i-1} to make only the necessary changes to compute weakly deciding vertices.

Similarly, the *compute*Γ procedure traverses the CFG backward from the set of given nodes SCC_{i-1} at the ith iteration of the ***while*** loop in Algorithm 7 and update $\gamma(n) = false$ for each visited node n. Thus, if $\gamma(n)$ is set to *false* at any earlier iteration j, it does not need to be visited at the later iterations. So, instead of calling *compute*Γ with the argument SCC_{i-1}, it is enough to call it with the argument Y_{i-1} in computing Γ.

Algorithms 8 and 9 facilitate computing minimal SCC incrementally. These algorithms are essentially similar to Algorithms 4 and 6 except that (i) A and γ are the input arguments of Algorithms 8 and 9 respectively, (ii) the initialization of A and γ are removed from these algorithms, and (iii) A is also the output of Algorithm 8; these modifications facilitate the incremental computation of minimal SCC. Finally, Algorithm 10 computes minimal SCC incrementally which initializes A and γ before the iterative computation of SCC in the ***while*** loop, and calls the *computeWDIncr* and the *compute*Γ*Incr* procedures with the argument Y instead of SCC unlike Algorithm 7. Moreover, $A(n) = n$ is set for all $n \in Y$ as Y is part of SCC (see Eq. (1)) and it may provide more weakly deciding vertices by the *computeWDIncr* procedure in the subsequent calls of the procedure. Even though the incremental computation of SCC does not improve the theoretical worst-case complexity, it improves the practical efficiency of the overall computation.

5.3. Worst-case complexity

Danicic et al. (2011) provided an algorithm to compute $\Gamma(G, N')$ which has the worst-case time complexity $O(|G|^2)$ where $|G| = |N| + |E|$. $O(|G|^2)$ is effectively $O(|N|^2)$ since the maximum out-degree of any CFG node is two. Moreover, their algorithm to compute minimal strongly control-closed superset of N' has the quartic worst-case cost, i.e., $O(|N|^4)$. In this section, we show that

Algorithm 10: computeMinSCCIncr

Input : $G = (N, E), N'$
Output: SCC

1 Apply Eq. (1) to initialize A
2 **forall** $(n \in N)$ **do** $\gamma(n) = true$
3 $SCC = N', Y = N'$
4 **while** $(true)$ **do**
5 $(A, WD) = computeWDIncr(G, Y, A)$
6 $(\Gamma, \gamma) = compute\Gamma Incr(G, Y, \gamma)$
7 $X = \{n : n \in WD \cup \Gamma \setminus SCC, \exists m \in succ(n).not(\gamma(m))\}$
8 $Y = \{n : n \in X, n$ is reachable from $SCC\}$
9 **if** $(Y == \emptyset)$ **then** ***break***
10 $SCC = SCC \cup Y$
11 **forall** $(n \in Y)$ **do** $A(n) = n$

$\Gamma(G, N')$ can be computed in linear time and minimal strongly control-closed superset of N' can be computed in cubic time in the worst-case in terms of the size of the CFG. Thus, we have improved the theoretical worst-case computational complexity of both algorithms by an order of $|N|$.

Lemma 9. *Given any CFG (N, E), the worst-case time complexity of Algorithm 6 is $O(|N|)$.*

Proof. The worst-case time complexity of Algorithm 6 is dominated by the ***while*** loop. The loop iterates as long as there exist elements in the worklist W. W cannot contain more than $|N|$ elements as (1) an element m can be inserted in W if $\gamma(m)$ is *true* (see the conditions of the ***if*** at line 7), and (2) once an element m is removed from W, $\gamma(m)$ is set to *false* (Line 5). Thus, no element is inserted in W twice. The ***forall*** loop in line 6 traverses all incoming edges to the CFG node n removed from W. Since each CFG node n can be inserted in W at most once, each incoming edge to n is visited at most once. The ***if*** condition at line 7 has constant cost since any CFG node has at most two successors according to Definition 1. All other operations in the ***while*** loop have constant cost. The iteration of the ***while*** and the ***forall*** loop (at line 6) are mutually dependent: removing a node n from W causes all edges (m, n) for $m \in pred(n)$ to be visited in the ***forall*** loop which in turn may add m to W. Thus, the maximum number of iterations of the ***while*** and the ***forall*** loops is $|N| + |E|$. Since all other operations have constant cost, the worst-case time complexity of the while loop, and hence Algorithm 6 is $O(|N|+|E|)$ which is effectively $O(|N|)$ since $|E| \leq 2 * |N|$. □

Theorem 5. *Given any CFG (N, E), the worst-case time complexity of Algorithm 7 is $O(|N|^3)$.*

Proof. In each iteration of the ***while*** loop in Algorithm 7, the set Y is computed and included in SCC, and SCC is excluded (line 5) in computing Y in the subsequent iterations. Thus, every iteration of the ***while*** loop computes a different $Y \subseteq N$, and since N is finite, the loop iterates at most $|N|$ times. The worst-case cost of each iteration is dominated by the *computeWD* procedure which is $O(|N|^2)$ according to Theorem 4. Thus, the worst-case cost of Algorithm 7 is $O(|N|^3)$. □

6. Experimental evaluation

We implemented ours and the weak and strong control closure algorithms of Danicic et al. (2011) in the Clang/LLVM compiler framework (Lattner and Adve, 2004) and run experiments in an Intel(R) Core(TM) i7-7567U CPU with 3.50 GHz and have 16 GB of RAM memory. In order to reduce biases to a particular

Table 4
Comparing execution times of WCC algorithms on selected benchmarks from SPEC CPU 2017 (Bucek et al., 2018).

#	Benchmarks	KLOC	#Proc	T_{wcc}	T_{wccD}	Speedup
1	Mcf	3	40	9.6	56.7	5.9
2	Nab	24	327	55.1	418.6	7.6
3	Xz	33	465	40.5	116.5	2.9
4	X264	96	1449	155.7	896.0	5.8
5	Imagick	259	2586	334.8	2268.9	6.8
6	Perlbench	362	2460	1523.3	32134.8	21.1
7	GCC	1304	17827	26658.1	634413.9	23.8
	Average Speedup = 10.6					

Table 5
Comparing execution times of SCC algorithms on selected benchmarks from SPEC CPU 2017 (Bucek et al., 2018).

Benchmarks	T_{incr}	T_{scc}	T_{sccD}	T_{scc}/T_{incr}	T_{sccD}/T_{incr}
Mcf	15.89	22.73	373.82	1.43	23.52
Xz	60.16	79.54	675.70	1.32	11.23
Nab	104.23	146.76	3561.19	1.41	34.17
X264	241.99	327.06	5898.98	1.35	24.38
Imagick	549.40	763.81	23198.71	1.39	42.23
Perlbench	2262.52	3176.54	513265.81	1.40	226.86
GCC	12699.26	17715.11	1412475.70	1.39	111.23

algorithm and mitigate any threat to validity, the implementation of all algorithms used smart data structures such as bitvectors and maps to improve its execution time and the code are optimized as much as possible. All implementations are released as open source under an open source license in a github repository.[1] The experiments are performed on a number of benchmarks consisting of approximately 2081 KLOC written in C language. In order to reduce any threat to validity, we utilize the same data structure.

Table 4 shows experimental results of computing WCC on seven benchmarks selected from the SPEC CPU 2017 (Bucek et al., 2018). The #Proc column indicates the number of procedures analyzed in the respective benchmarks, T_{wcc} and T_{wccD} columns show total runtime of the WCC algorithms of ours and Danicic et al. and the Speedup column indicates the speedup of our approach over Danicic et al. which is calculated as T_{wccD}/T_{wcc}. Each procedure is analyzed 10 times and the N'-sets are chosen randomly for each run. All times are recorded in microseconds which are converted to milliseconds and the analysis times reported in Table 4 are the average of 10 runs.

Regarding the correctness, both algorithms compute the same weakly control closed sets. As shown in Table 4, we obtain the highest and the lowest speedup of 23.8 and 2.9 from the *GCC* and the *Xz* benchmarks, and an average speedup from all benchmarks is 10.6. The *Xz* benchmark provides the lowest speedup due to the fact that it has fewer procedures than *GCC* and the sizes of the CFGs for most procedures in this benchmark are very small; the average size of a CFG (i.e. number of CFG nodes) is only 8 per procedure. On the other hand, *GCC* has 38 times more procedures than *Xz* and the average size of a CFG per procedure is 20. Also, the greater speedups are obtained in larger CFGs. There are 171 and 55 procedures in *GCC* with the sizes of the CFGs greater than 200 and 500 respectively, and the maximum CFG size is 15912 whereas the maximum CFG size in *Xz* is 87. For benchmarks like *Mcf* and *Nab*, even though they have fewer procedures than *Xz*, the average CFG size per procedure in these benchmarks is 21 and 16 respectively.

Since Algorithm 2 and 3 dominate the computational complexity of computing WCC, we compare the execution times of these algorithms in Fig. 5. We also plotted the functions $N \log N$ and N^2 to compare the execution curves of the algorithms with these functions. All times are measured in microseconds and an average of 10 runs. If there exist several CFGs with the same size, we keep the execution time of only one of them. As illustrated in the figure, Algorithm 2 performs consistently. However, the performance of Algorithm 3 varies above or below the performance of Algorithm 2. This due to the fact that it shows optimal performance when *noDisjointNPath* procedure is called minimally. The performance curves of both algorithms are closer to the $N \log N$ curve for perlbench and gcc benchmarks and closer to the linear curve for other benchmarks depicted in Fig. 5 when the times are measured in microseconds.

[1] https://github.com/anm-spa/CDA

We also have evaluated our algorithms by performing the same experiments on a virtual machine (VM) running on the real machine as specified above. The virtual machine uses a 64-bit Ubuntu OS with 10 GB RAM having 2 cores and the real machine runs Mac OS Version 10.15.4 with 16 GB RAM. Due to randomization, the experiments have different N' sets. We obtain a maximum speedup of 12 for *Perlbench* and an average speedup of 5.7 on all benchmarks from the experiments on the VM. Even though we obtain a smaller speedup compared to the speedup on the real machine, our algorithm is still several times faster than the WCC computation of Danicic et al. and we obtain similar performance curves for all benchmarks on VM. Evidently, our algorithm improves the state-of-the-art computation of weak control closure by an order of magnitude.

Table 5 shows experimental results of computing SCC on the selected benchmarks from the SPEC CPU 2017 (Bucek et al., 2018). The symbols T_{incr}, T_{scc}, and T_{sccD} denote the execution time of incremental computation of SCC by Algorithm 10, the SCC computation by Algorithm 7, and the SCC computation of Danicic et al. (2011) respectively. All times are recorded in microseconds which are converted to milliseconds and the reported analysis times in Table 5 are the average of 10 runs. The fifth column lists the speedup of the incremental computation of SCC over the SCC computation by Algorithm 7 which is calculated as T_{scc}/T_{incr}. The final column lists the speedup of the incremental computation of SCC by Algorithm 10 over the SCC computation of Danicic et al. which is calculated as T_{sccD}/T_{incr}. Each procedure is analyzed 10 times and the N'-sets are chosen randomly for each run.

Regarding the correctness, all SCC computation produce exactly the same strongly control-closed sets. Regarding the execution time, the incremental SCC computation by Algorithm 10 is the fastest among the three SCC algorithms in all benchmarks. On an average, it is 1.38 times faster than the iterative computation of SCC computation by Algorithm 7. However, both Algorithms 10 and 7 are significantly faster than the algorithm of Danicic et al. We obtain the minimum speedup of 11.23 in the *Xz* benchmark, the maximum speedup of 226.86 on the *Perlbench* benchmark, and the average speedup of 67.66 on each benchmarks by Algorithm 10 over Danicic et al. Thus, we have improved not only the theoretical worst-case complexity but also the practical efficiency of computing SCC by an order of magnitude. Moreover, the execution times in Tables 4 and 5 confirm that computing WCC is more efficient than computing SCC.

7. Related work

Denning and Denning (1977) are the pioneers to use dominator-based approach to identify program instructions influenced by the conditional instructions in the context of information-flow security. Weiser (1981), the pioneer and prominent author in program slicing, used their approach in program slicing. However, the first formal definition of control dependence is provided by Ferrante et al. (1987) in developing the program dependence graph (PDG) which is being used for program slicing and program

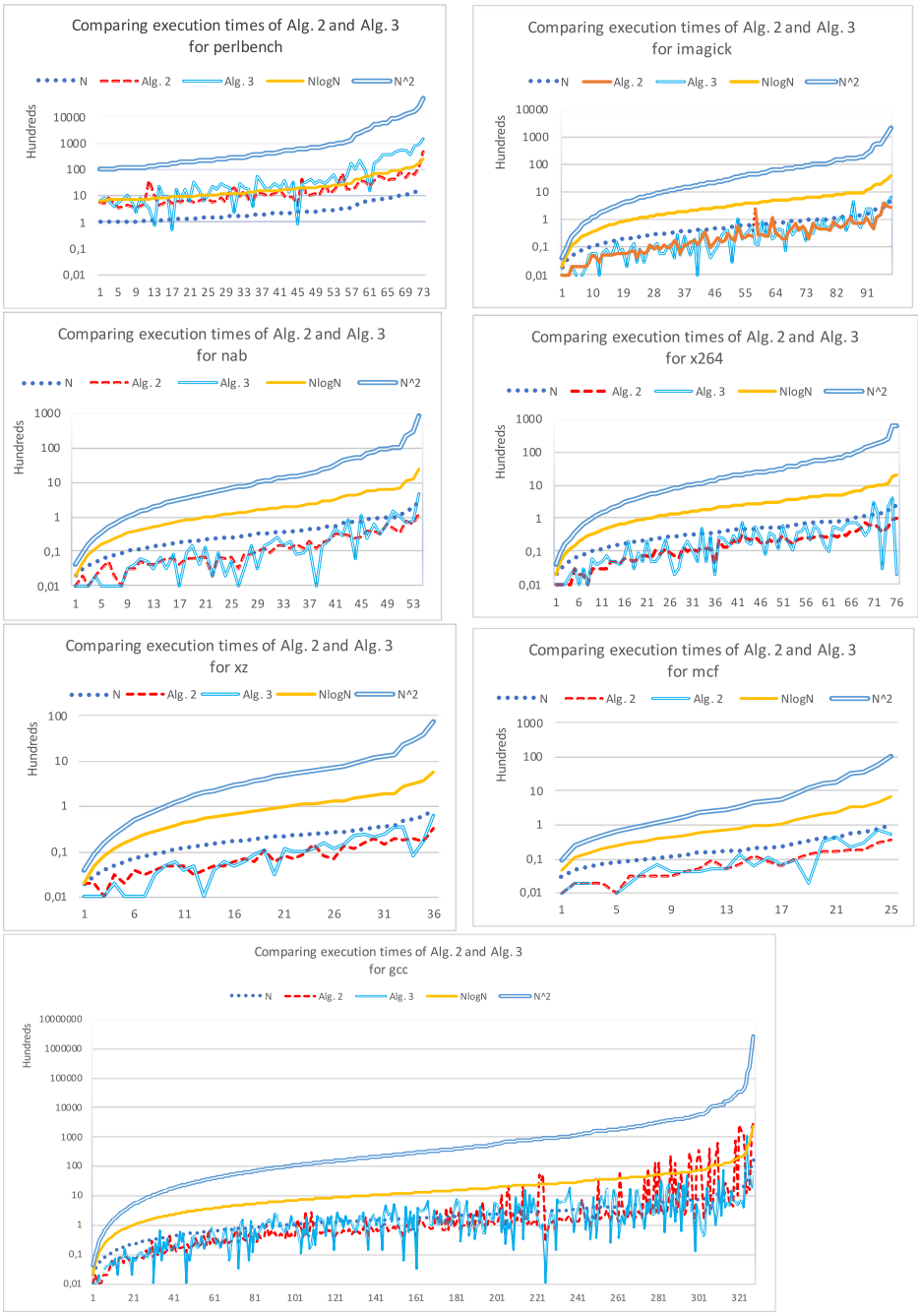


Fig. 5. Comparing execution times of Algorithms 2 and 3. Execution time curves are also compared with the $N \log N$ and N^2 functions where N represents the number of nodes in the CFG. X-axis represents selected CFGs from the respective benchmarks. Y-axis represents either the execution times of the algorithms measured in microseconds or the value of $N \log N$ and N^2. All charts are displayed in the logarithmic scale.

optimization. This definition became standard afterward and is being used for over two decades.

Podgurski and Clarke (1990) provided two control dependence relations called weak and strong syntactic dependence. The strong control dependence corresponds to the standard control dependence relation. The weak control dependence subsumes the strong control dependence relation in the sense that any strong control dependence relation is also a weak control dependence. Moreover, the weak control dependence relation is nontermination sensitive. Bilardi and Pingali (1996) provided a generalized framework for the standard and the weak control dependence relation of Podgurski and Clarke by means of the dominance relation parameterized with respect to a set of CFG paths. Different classes of CFG path set provides different control dependence relations.

Ranganath et al. (2007, 2005) considered CFGs possibly having multiple end nodes or no end node. These kinds of CFGs originate from programs containing modern program instructions like exceptions or nonterminating constructs often found in web services or distributed systems. They also considered low-level code such as JVM producing irreducible CFGs, and defined a number of control dependency relations that are nontermination (in)sensitive and conservatively extend the standard control dependency relation. The worst-case time complexity of the algorithms for computing their control dependences is $O(|N|^4 log|N|)$ where $|N|$ is the number of vertices of the CFG.

The control dependence relations defined later are progressively generalized than the earlier definitions, but one may be baffled by the overwhelming number of such definitions, e.g. in Ranganath et al. (2007), to choose the right one. Danicic et al. (2011) unified all previously defined control dependence relations and provided the most generalized non-termination insensitive and nontermination sensitive control dependence called weak and strong control-closure. These definitions are based on the weak and strong projections which are the underlying semantics for control dependence developed by the authors. These semantics are opposite to that of Podgurski and Clark in the sense that Danicic et al.'s weak (resp. strong) relation is similar to Podgurski and Clark's strong (resp. weak) relation. The worst-case time complexity of their weak and strong control closure algorithms are $O(|N|^3)$ and $O(|N|^4)$ where $|N|$ is the number of vertices of the CFG. Léchenet et al. (2018) provided automated proof of correctness in the Coq proof assistant for the weak control closure algorithm of Danicic et al. and presented an efficient algorithm to compute such control closure. The efficiency of their method is demonstrated by experimental evaluation. However, no complexity analysis of their algorithm is provided.

Recently, Chalupa et al. (2021) have provided improved algorithms to compute the non-termination sensitive control dependence (NTSCD) and decisive order dependence (DOD) of Ranganath et al. (2007, 2005). The asymptotic complexity of their algorithms are $O(|N|^2)$ and $O(|N|^3)$ for computing NTSCD and DOD respectively where $|N|$ is the number of vertices of the CFG. However, Danicic et al. (2011) have shown that a subset N' of the set of CFG nodes is strongly control-closed in the CFG if N' is closed under both the NTSCD and DOD relations when the Start node of the CFG is in N'. Therefore, their algorithms can compute the SCC with the restriction that the Start node must belong to N'.

Khanfar et al. (2019) developed an algorithm to compute all direct control dependencies to a particular program statement for using it in demand-driven slicing. Their method only works for programs that must have a unique exit point. Neither the computational complexity nor the practical performance benefits of their algorithm are stated. On the other hand, we compute minimal weak and strong control closure for programs that do not have such restrictions. Our method improves the theoretical computational complexity of computing weak and strong control closure than the state-of-the-art methods, and it is also practically efficient.

Recently, we have extended the definitions of weak and strong control closure for interprocedural programs (Masud and Lisper, 2021) in proving the correctness of dependency-based program slicing (Khanfar et al., 2015; Lisper et al., 2015). Moreover, we have shown that the standard dominance frontier-based SSA construction method increases the size of the SSA program by computing a significant amount of unnecessary ϕ functions (Masud and Ciccozzi, 2019, 2020), and provided an algorithm to generate minimal SSA programs. We have a work-in-progress that shows that placing ϕ functions in SSA computation and obtaining weakly deciding vertices are dual problems. We have developed an algorithm that can compute both the weakly deciding vertices and the placements of ϕ-functions which can be used as an efficient and generalized alternative in computing minimal SSA programs.

8. Conclusion and future work

Danicic et al. provided two generalizations called weak and strong control closure (WCC and SCC) that subsume all existing nontermination insensitive and nontermination sensitive control dependency relations. However, their algorithms to compute these relations have cubic and quartic worst-case complexity in terms of the size of the CFG which is not acceptable for client applications of WCC and/or SCC such as program slicing. In this paper, we have developed an efficient and easy to understand method of computing minimal WCC and SCC. We provided the theoretical correctness of our method. Our algorithms to compute WCC and SCC have the quadratic and the cubic worst-case time complexity in terms of the size of the CFG. We experimentally evaluated the algorithms for computing WCC and SCC of ours and Danicic et al. on practical benchmarks. For the WCC, we obtained the highest 23.8 and on average 10.6 speedups compared to the state-of-the-art method. For the SCC, the highest and the average speedups are 226.86 and 67.66 compared to the method of Danicic et al. The performance of our WCC algorithm for practical applications is closer to either NlogN or linear curve in most cases when time is measured in microseconds. Thus we improve the practical performance of WCC and SCC computation by an order of magnitude.

The future extension of this work will be to develop methods for computing WCC and SCC for interprocedural programs and apply WCC and SCC in different dependence-based program analyses.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This research is supported by the Swedish Knowledge Foundation, Sweden (KKS) through the HERO project.

References

Amtoft, T., 2007. Correctness of Practical Slicing for Modern Program Structures. Department of Computing and Information Sciences, Kansas State University.

Bilardi, G., Pingali, K., 1996. A framework for generalized control dependence. SIGPLAN Not. 31 (5), 291–300. http://dx.doi.org/10.1145/249069.231435.

Bucek, J., Lange, K.-D., v. Kistowski, J., 2018. SPEC CPU2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. ICPE '18, ACM, New York, NY, USA, pp. 41–42. http://dx.doi.org/10.1145/3185768.3185771.

Chalupa, M., Klaska, D., Strejček, J., Tomovic, L., 2021. Fast computation of strong control dependencies. In: Silva, A., Leino, K.R.M. (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 887–910.

Danicic, S., Barraclough, R., Harman, M., Howroyd, J.D., Kiss, Á., Laurence, M., 2011. A unifying theory of control dependence and its application to arbitrary program structures. Theoret. Comput. Sci. 412 (49), 6809–6842.

Denning, D.E., Denning, P.J., 1977. Certification of programs for secure information flow. Commun. ACM 20 (7), 504–513. http://dx.doi.org/10.1145/359636.359712.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9 (3), 319–349. http://dx.doi.org/10.1145/24039.24041.

Khanfar, H., Lisper, B., Masud, A.N., 2015. Static backward program slicing for safety-critical systems. In: de la Puente, J.A., Vardanega, T. (Eds.), Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings. In: Lecture Notes in Computer Science, vol. 9111, Springer, pp. 50–65. http://dx.doi.org/10.1007/978-3-319-19584-1_4.

Khanfar, H., Lisper, B., Mubeen, S., 2019. Demand-Driven Static Backward Slicing for Unstructured Programs. Mälardalen Real-Time Research Centre, Mälardalen University, URL http://www.es.mdh.se/publications/5511-.

Lattner, C., Adve, V., 2004. The LLVM compiler framework and infrastructure tutorial. In: LCPC'04 Mini Workshop on Compiler Research Infrastructures. West Lafayette, Indiana.

Léchenet, J.-C., Kosmatov, N., Le Gall, P., 2018. Fast computation of arbitrary control dependencies. In: Russo, A., Schürr, A. (Eds.), Fundamental Approaches to Software Engineering. Springer International Publishing, Cham, pp. 207–224.

Lisper, B., Masud, A.N., Khanfar, H., 2015. Static backward demand-driven slicing. In: Asai, K., Sagonas, K. (Eds.), Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015. ACM, pp. 115–126. http://dx.doi.org/10.1145/2678015.2682538.

Masud, A.N., 2020. Simple and efficient computation of minimal weak control closure. In: Pichardie, David, Sighireanu, Mihaela (Eds.), Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings. 12389, Springer International Publishing, Cham, pp. 200–222. http://dx.doi.org/10.1007/978-3-030-65474-0_10.

Masud, A.N., Ciccozzi, F., 2019. Towards constructing the SSA form using reaching definitions over dominance frontiers. In: 19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019. IEEE, pp. 23–33. http://dx.doi.org/10.1109/SCAM.2019.00012.

Masud, A.N., Ciccozzi, F., 2020. More precise construction of static single assignment programs using reaching definitions. J. Syst. Softw. 166, 110590. http://dx.doi.org/10.1016/j.jss.2020.110590.

Masud, A.N., Lisper, B., 2021. Semantic correctness of dependence-based slicing for interprocedural, possibly nonterminating programs. ACM Trans. Program. Lang. Syst. 42 (4), 19:1–19:56. http://dx.doi.org/10.1145/3434489, URL https://doi.org/10.1145/3434489.

Ottenstein, K.J., Ottenstein, L.M., 1984. The program dependence graph in a software development environment. SIGSOFT Softw. Eng. Notes 9 (3), 177–184. http://dx.doi.org/10.1145/390010.808263.

Pingali, K., Bilardi, G., 1997. Optimal control dependence computation and the roman chariots problem. ACM Trans. Program. Lang. Syst. 19 (3), 462–491.

Podgurski, A., Clarke, L.A., 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng. 16 (9), 965–979.

Prosser, R.T., 1959. Applications of Boolean matrices to the analysis of flow diagrams. In: Papers Presented At the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference. IRE-AIEE-ACM '59 (Eastern), ACM, New York, NY, USA, pp. 133–138.

Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B., 2005. A new foundation for control dependence and slicing for modern program structures. In: European Symposium on Programming. In: LNCS, vol. 3444, Springer-Verlag, pp. 77–93.

Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B., 2007. A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. 29 (5).

Weiser, M., 1981. Program slicing. In: Proc. 5th International Conference on Software Engineering. ICSE '81, IEEE Press, Piscataway, NJ, USA, pp. 439–449, URL http://dl.acm.org/citation.cfm?id=800078.802557.