Fast and incremental computation of weak control closure

Abu Naser Masud[0000−0002−4872−1208]

School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
abu.naser.masud@mdu.se

Abstract. Control dependence is a fundamental concept used in many program analysis techniques such as program slicing, program debugging, program parallelization, and detecting security leaks. Since the introduction of this concept in the late eighties, numerous definitions of control dependencies and their computation methods have been developed. The later definitions are progressively more generalized covering a wide spectrum of modern programming language constructs. The most generalized concepts are the weak and strong control closure (WCC and SCC) that capture the nontermination (in)sensitive control dependencies of a given program. In this paper, we have developed a novel method to compute WCC incrementally. Any client application of WCC such as program slicing requires computing the WCC repeatedly in a fixpoint computation. An incremental algorithm to compute WCC will improve the performance of the client application significantly. We have provided the proof of correctness and the theoretical worst-case complexity of our algorithm. We have performed an experimental evaluation on well-known benchmarks, and our experiments reveal that we have significantly improved the practical efficiency in computing WCC incrementally. We have obtained an average speedup of 31.03 in all benchmarks and a maximum speedup of 35.29 than the best state-of-the-art algorithm computing WCC.

Keywords: Control Dependency · weak control closure · strong control closure · program slicing · nontermination insensitive · nontermination sensitive

1 Introduction

Control dependence is a fundamental concept used in many program analyses techniques such as compiler optimization and debugging [6], program slicing [22,7,11], and information flow security [5]. It expresses a relation between two program instructions stating that one controls the execution of another. The state-of-the-art control dependence computation is based on computing postdominator relations [6]. It is the fastest algorithm which is applicable for programs that must exit from a single program point. However, modern software systems such as nonterminating reactive systems or distributed web ser-

vices do not have any exit point and postdominator-based control dependence computation algorithms are not applicable for such systems [20,1].

Danicic et al. [4] introduced the concept of weak and strong control closure (WCC and SCC) that capture weak and strong forms of control dependencies from (non)terminating systems that are nontermination insensitive and nontermination sensitive respectively. These are the most general form of the closure relation of control dependencies. Program transformation technique such as program slicing may require preserving nontermination in the sliced program if the original program is nonterminating. Slicing based on SCC preserves nontermination and produces larger slices which may be useful for program verification. On the other hand, slicing based on WCC may not preserve nontermination and produces smaller slices. There exist numerous definitions of control dependencies in the literature [22,6,19,2,21]. All such dependencies are the special cases of control dependencies captured by WCC and SCC. Thus, WCC and SCC have a wider applicability than the control dependencies based on computing postdominator relations.

Let us illustrate these concepts and their relations by the program P in Fig. 1. Suppose we are interested to detect program statements that affect the value of x at Statement 8. The assignment to x at Statements 1 and 4 directly affect the value of x at Statement 8. We can obtain these direct influences by computing data dependencies. Moreover, the boolean expression $i > 0$ at Statement 2 indirectly affects the value of x at Statement 8 as it decides whether Statement 4 will be executed or not. We can obtain these indirect influences by computing control dependencies. Furthermore, Statement 6 directly affects the boolean expression at Statement 2 due to data dependence, and thus statements 1, 2, 4, and 6 affect the value of x at Statement 8. Statement 4 is control dependent on Statement 2 as it decides whether statement 4 will be executed or not. The standard control dependence computation method identifies this control dependence as follows: there exist two paths from Statement 2 to the end of this program in which Statement 4 always executes in one path and Statement 4 may not execute in the other path. This proves that Statement 2 controls the execution of Statement 4. This method only works if the program has a single exit point. However, if the whole program is under an infinite loop such as the following:

```
1: x=5;
2: while(i>0)
3: {
4:    x=x+i;
5:    y=y−i;
6:    i=i−1;
7: }
8: print(x)
```

Fig. 1: Programs P

$$\textit{Program Q: } \textbf{while}(\textit{true})\{S_1, \ldots, S_8\}$$

where $S_1, \ldots, S_8$ refers to statements 1 to 8 in Program P, then the above method for computing the control dependencies will not work since there is no exit point to this code. WCC (see Def. 4 in Sec. 2) is the generalization of the standard control dependencies that also works for nonterminating programs and computes the closure relation of control dependencies. For example, if we would like to

know what affects the computation of x at Statement 8 in Program Q, we see that there are two paths from Statement 2 to statements 4 and 8. This is enough to say that a control closure of a set S of program statements including statements 8 and 4 must include Statement 2 (provided that Statement 2 is reachable from S) as it decides which one (Statement 4 or Statement 8) will be executed next. One difference between SCC and WCC is that the SCC of S will also include the ***while****(true)* statement as it preserves the nontermination. Statement 2 is nontermination sensitively control dependent on the ***while****(true)* statement and thus SCC will include this statement.

1 Let C be the the slicing criterion, and let $S = C$.
2 **repeat**
3 $\quad S' := \bigcup_{n \in S} \{m : m \overset{dd}{\rightarrow}^{*} n\}$
4 $\quad S := cl(S')$
5 **until** $S = S'$

Algorithm 1: SLICING

Danicic et al.'s original algorithms to compute WCC and SCC are expensive. Most recent works [12,14,18] on computing WCC and SCC have shown performance improvements in these algorithms. These improvements are mostly on the one-time application of these algorithms. However, client applications such as program slicing require computing WCC/SCC incrementally. Existing algorithms lose performance due to repeated computation of unnecessary information. To illustrate this fact, we recall Alg. 1 from [12,14][1] which is the static backward slicing algorithm computing the slice set S from the given slicing criterion C. Given any control flow graph (CFG) representation of a program and the slicing criterion C which is a subset of the CFG nodes, Alg. 1 computes S until a fixpoint is reached. The relation $\overset{dd}{\rightarrow}^{*}$ denotes the transitive-reflexive closure of the data dependence relation $\overset{dd}{\rightarrow}$. The function $cl(.)$ computes WCC or SCC in each fixpoint iteration. It is obvious that an incremental computation of this function will improve the overall performance of the program slicing algorithm.

In this paper, we have developed a novel algorithm that is able to compute WCC incrementally. We proved the correctness of our algorithm theoretically and experimentally, provided the theoretical worst-case time complexity of our method which is quadratic in the size of the CFG, implemented our algorithm in the Clang/LLVM compiler framework, and compared our results with the best state-of-the-art method by performing experimental evaluation on SPEC CPU 2017 benchmarks. Our experiments reveal that the algorithm developed in this paper is the fastest among all algorithms computing WCC reported in the literature if WCC needs to be computed incrementally. We have obtained the maximum speedup of 35.29 on our largest benchmark and an average speedup of

[1] We replaced the ***goto*** statement in [12,14] by the ***repeat***..***until*** *loop*.

31.03 on all benchmarks with respect to the best baseline approach computing WCC.

The remainder of this paper is organized as follows. Sec. 2 brings some relevant concepts and notations from the literature on control dependency and WCC, Sec. 3 provides the details of our algorithm developed in this paper, the proof of correctness, and its theoretical worst-case time complexity, Sec. 4 explains the experimental evaluation, Sec. 5 discusses the works that are related to ours, and Sec. 6 concludes the paper.

2 Background

In this section, we recall definitions of CFG, control dependency, WCC, and other related concepts from the relevant literature [14,4]. The definitions of Control dependency and WCC are provided at the level of CFG representation of programs. First, we recall the formal definition of a control flow graph (CFG) from our earlier study [12,14].

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 has at most one true successor and at most one false successor, and all other nodes have at most one successor.*

Applications like program slicing may remove part of the code and we may obtain a CFG from such code in which a Cond node has either or both of the successors missing. Other kinds of nodes may have a missing successor as well. An execution that reaches such nodes may be silently nonterminating because an execution may not proceed and the control is not returned to the operating system. Moreover, a CFG may not have an End node and the execution of its code may possibly be nonterminating. Thus, our definition of CFG can represent a wide range of practical programs. Note that our CFGs are deterministic according to the definition.

The functions $succ_G(n)$ and $pred_G(n)$ denote the set of successors and predecessors of the CFG node n in the graph G. We sometimes drop the subscript G if it is understood from the context. A path in a graph G is the sequence of CFG nodes $n_1, \ldots, n_k$ such that $n_{i+1} \in succ_G(n_i)$ (also $n_i \in pred_G(n_{i+1})$) for all $1 \leq i \leq k-1$. We use the notation $[n_1..n_k]$ to denote such a path. A *non-trivial* path contains more than one node; otherwise, the path is a *trivial* path. A path is *proper* if its initial and final vertices are distinct. Two paths $[n_1..n_k]$ and $[m_1..m_l]$ are *disjoint* if $n_i \neq m_j$ for any $k, l > 0$, $1 \leq i \leq k$, and $1 \leq j \leq l$. If $[n_2..n_k]$ and $[m_2..m_l]$ are *disjoint* paths and $n_1 = m_1$, then we say that $[n_1..n_k]$ and $[m_1..m_l]$ are *disjoint* paths from n_1. We use the notation $n \in \pi$ to denote

that the CFG node n belongs to the path π, and use the notation $n \in \pi - S$ to indicate that n is any node in the path π that does not belong to the set of CFG nodes S.

We now recall the definition of WCC from Danicic et al. [4] with some auxiliary relevant definitions. In what follows, let $G = (N, E)$ be a CFG, and let $N' \subseteq N$.

Definition 2 (N'-Path). *An N'-path is a finite path $[n_1..n_k]$ in a CFG G where $k > 1, n_k \in N'$ and $n_i \notin N'$ for all $1 < i < k$.*

Intuitively, an N'-path must end at a node in N', the first node in this path can be any node from N (or N'), and no intermediate node in this path must be from N'. With this definition, we can now define an N'-weakly committing node as follows:

Definition 3 (N'-weakly committing). *A node $n \in N$ is N'-weakly committing in a CFG 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.*

The following definition states whether a given subset of CFG nodes N' is N'-weakly control-closed.

Definition 4 (Weak control closure). *N' is weakly control-closed in G if and only if all nodes $n \notin N'$ that are reachable from N' are N'-weakly committing in G.*

Given any subset N' of CFG nodes, if N' is not weakly control-closed in the CFG G according to Def. 4, then we compute the WCC set of N' denoted $WCC(N')$ (or WCC for brevity) such that $N' \subseteq WCC(N') \subseteq N$. The definition of weak control closure captures the control dependencies obtained from postdominator relations. To illustrate this relation, we now bring the definition of postdominator relation and the control dependencies based on this relation.

A CFG node n *postdominates* another CFG node m if and only if every path from m to the *End* node must go through n. If $n \neq m$ in this definition, then we say that n *strictly postdominates* m. Note that this definition relies on the fact that the CFG must have a unique End node. The standard definition of postdominator-based control dependencies was first introduced by Ferrante et al. [6] as follows:

Definition 5 (Control Dependency [6,21,12]). *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.*

Since the definition of postdominator relies on the existence of a unique *End* node, the above definition of control dependence is applicable when this restriction holds. Intuitively, the relation $m \xrightarrow{cd} n$ holds when there exist two branches of m such that n is always executed in one branch and may not execute in the other branch. To illustrate this relationship with WCC, let us assume that N'

includes n, the unique End node n_e, and the Start node $n_\triangleright$. Also, assume that all nodes are reachable from node $n_\triangleright$. Then, $[m..n]$ is an N'-path. Since m is not strictly postdominated by n, there must be another path $[m..n_e]$ which is also an N'-path. Then, node m is not N'-weakly committing due to having two N'-paths with different endpoint. Thus, N' is not a WCC due to not capturing the control dependency relation $m \xrightarrow{cd} n$, and a WCC of N' must include m.

The concept of N'-weakly deciding nodes are introduced by Danicic et al. [4] to provide an algorithm to compute the WCC of N'.

Definition 6 (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 such 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 a subset $N' \subseteq N$ of CFG nodes in a CFG G can be computed according to the following equation:

$$WCC(N') = N' \cup \{n : n \in WD_G(N'), n \text{ is reachable from } N' \text{ in } G\} \quad (1)$$

Example 1. Consider the CFG in Fig. 2. Let $N' = \{n_0, n_1, n_5, n_{11}\}$. Here are some examples of N'-paths in this CFG:

$$\begin{aligned}
\pi_0 &= n_1, n_0 \\
\pi_1 &= n_3, n_1 \\
\pi_2 &= n_3, n_2, n_1 \\
\pi_3 &= n_6, n_4, n_3, n_1 \\
\pi_4 &= n_6, n_5 \\
\pi_5 &= n_{10}, n_1 \\
\pi_6 &= n_{10}, n_9, n_6, n_4, n_3, n_2
\end{aligned}$$

The path $[n_3..n_0]$ is not an N'-path as it includes the node $n_1 \in N'$. The CFG node n_1 is N'-weakly committing since there exists a single N'-path π_0. Also, node n_3 is N'-weakly committing as both π_1 and π_2 have the same endpoint n_1. However, node n_6 is not N'-weakly committing due to the paths π_3 and π_4. It is an N'-weakly deciding node. Thus, N' is not weakly control closed. Similarly, nodes $n_7, \ldots, n_{10}$ are N'-weakly deciding due to having two disjoint N'-paths from these nodes. As all these nodes are reachable from $n_{11} \in N'$, the WCC of N' must include these nodes according to Eq. 1.

Regarding the program semantics of the client application of WCC or SCC such as program slicing or information flow security, the execution semantics of programs are captured/preserved through computing the additional dependencies such as data dependencies as shown in Alg. 1. However, the data dependencies are not enough to capture the indirect influences of conditional statements or boolean instructions in loop statements. Thus, WCC/SCC is applied on top of data dependencies to capture these indirect influences.

3 Incremental computation of WCC

We compute WCC in two steps. First, we generate an *influencer graph* $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ from the CFG $G = (N, E)$ such that $\mathcal{N} = N$, and $\mathcal{G}$ is a directed graph that encodes direct influences to all CFG nodes by the Cond nodes. We say that a Cond node n influences the execution of a node m if there exists a path $[n..m]$ in the CFG G which does not include any other Cond node. The *influencer graph* $\mathcal{G}$ is thus a program representation encoding all direct influences in the source code. This simplistic informal definition of influence is extended in the following section to construct the influencer graph $\mathcal{G}$. In the second step, given any set $N' \subseteq N$, we traverse $\mathcal{G}$ from the nodes in N' to detect all Cond nodes that are weakly deciding and are reachable from the nodes in N' in G. We compute the graph $\mathcal{G}$ only once. But, we traverse it incrementally to compute WCC and avoid recomputing decisions taken earlier.

3.1 Generating the influencer graph

The intuitive idea of having an influencer graph is the following. Given any CFG node n, we should immediately recognize which Cond node m controls the execution of n (if any). Usually, m is the last Cond node in any path from the Start node to n. Subsequently, there may be other Cond nodes preceding m that control the execution of m. Let π be any path from the Start node to n, and let $m_1, \ldots, m_k$ be the subsequence of Cond nodes in π. We intend to obtain the edges (n, m_k) and (m_{i+1}, m_i) for all $1 \leq i \leq k-1$ in the influencer graph. Then, given any CFG node $n \in N'$, any path $[n..m_i]$ in the influencer graph indicates that there may be an N'-path $[m_i..n]$ in the CFG if no node in this path is in N' except n. After obtaining the influencer graph of any CFG, we can limit our search space to verify which Cond nodes should belong to the WCC. We perform the search in the influencer graph in a way, as explained in the next section, that ensures that the paths in the influencer graph correspond to N'-paths in the CFG. Since any Cond node m which is N'-weakly deciding and thus has two disjoint N'-paths may belong to the WCC of N', we assign each edge in the influence graph by a branch number to identify the disjointness of the N'-paths. In order to generate the influencer graph, we consider a distinct branch number for each branch of a Cond node. A CFG node is called a Join node if it has multiple predecessors. A Join node is also a Cond node when it has two successors. We assign a default label to the branch emerging from a Join but nonCond node. The influencer graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is a directed graph with edge labels that can be obtained from the CFG $G = (N, E)$. Any edge $(n, m, l) \in \mathcal{E}$ in $\mathcal{G}$ implies that there exists a path $[m..n]$ in the CFG G, m is a Cond or a Join node, and the edge label l can take any value from the set $\{0, 1, 2, 3\}$ to represent the branch number. The semantics of the encoding of edge label l is the following:

- l=0 represents the default branch number emerging from the Join node m,

- $l = 1$ or $l = 2$ implies that m is a Cond node which usually has at most two branches and n can be reached from m in the CFG by traversing the branch marked by l,
- $l = 3$ implies that m is a Cond node and n can be reached from m by traversing any of the two branches of m.

Since any CFG node can have at most two branches according to the definition of CFG (Def. 1), we need at most four unique branch numbers. In the practical implementation of our approach, we handle more than four branch labels to handle ***switch*** statements. However, we restrict ourselves to four unique branch numbers for the brevity of our presentation.

In the following, we provide the formal definition of an influencer graph $\mathcal{G}$.

Definition 7 (Influencer graph). *Let $G = (N, E)$ be a CFG. An influencer graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ of a CFG G consists of the set of CFG nodes $\mathcal{N} = N$, and the set of edges $\mathcal{E}$ containing all edges (n, m, l) such that m is either a Join or a Cond node, and there exists a CFG path $[m..n]$ such that no node $n' \in [m..n] - \{n, m\}$ is a Cond or a Join node. The edge label l can take any value as follows:*

1. *$l = 0$ if m is a Join node*
2. *$l = 3$ if m is a Cond node, n is a Join node, and no disjoint path exists from m to n in the CFG, and*
3. *$l \in \{1, 2\}$ if m is a Cond node, n is not a Join node, and for all edges $(n, m', l') \in \mathcal{E}$ such that $[m..n]$ and $[m'..n]$ are disjoint paths in G, we must have that $l' \neq l$.*

Any edge (n, m, l) in $\mathcal{G}$ implies that m may affect the execution of n if $l \neq 0$. Thus, m may influence the execution of n, and the graph $\mathcal{G}$ encodes all such influences in the CFG G. When $l = 0$, m will not directly influence n, but n may be influenced by another node m' if there exists an edge (m, m', l') in $\mathcal{G}$ such that $l' \neq 0$. In fact, for all sequences of edges $(n, m, 0)$ and (m, m', l') in $\mathcal{G}$, we can remove the edge $(n, m, 0)$ and add a new edge (n, m', l') in $\mathcal{G}$ in a post-processing phase of generating $\mathcal{G}$. This compact representation will only encode all direct influences in $\mathcal{G}$. However, we consider it as a syntactic sugar and keep the edge $(n, m, 0)$ in $\mathcal{G}$ to simplify its generation. Since any CFG node has at most two successors according to the definition of CFG, we need at most four unique labels for the influencer graph $\mathcal{G}$.

Fig. 2(a) presents the CFG of a function taken from the Perlebench benchmark obtained from the well-known SPEC CPU2017 [3] benchmark suite. This CFG is generated by the Clang frontend [9], and each node in this CFG represents a basic block containing the straight-line sequence of instructions written in C language. All Cond nodes (e.g., n_9, n_8, n_3, etc.) in this CFG have two successors, all Join nodes (e.g., n_1, n_6, n_3, etc.) have multiple predecessors, and there exist nodes that are both Cond and Join nodes (e.g. n_6, n_3, etc.). Fig. 2(b) presents the influencer graph that is obtained from the CFG in (a) according to Def. 7.

Alg. 2 generates the influencer graph $\mathcal{G}$ of a given CFG G. It uses the following notations/functions:

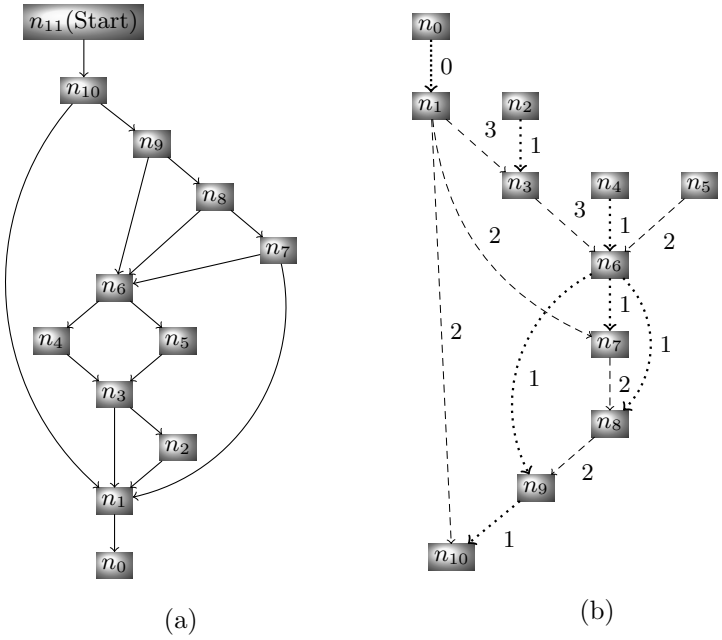

Fig. 2: (a) The CFG generated by the Clang frontend for the "Perl_do_open_raw" function taken from the Perlbench benchmark in SPEC CPU2017 [3] (we omit the program instructions for simplicity), (b) The *influencer graph* $\mathcal{G}$ generated by Alg. 2 in which edge styles are differed by their edge labels

- The function $\mathcal{I} : N \to N \cup \{\bot, \iota\}$ records the direct influence of a given CFG node. For example, $\mathcal{I}(n) = m$ implies that m may be a Cond node (or a Join node) which may influence the execution of n directly (resp. indirectly). Initially, $\mathcal{I}(n) = \bot$ for all CFG nodes n representing that an influencer node of n is yet to be recognized. Also, we initially set $\mathcal{I}(n_\triangleright) = \iota$ to denote that the execution of the Start node $n_\triangleright$ is not influenced by any other node.
- The function $\mathcal{E}_m$ denotes the set of all edges from the node m in the influencer graph $\mathcal{G}$. In particular, any $(n, l) \in \mathcal{E}_m$ implies that (m, n, l) is an edge in $\mathcal{G}$.
- The function $\mathcal{D}ist(n, m)$ denotes the minimum distance of node n from either m if m is a Cond node or a Cond node immediately before m in the CFG if m is a Join node. We compute this function during the generation of the influencer graph and use this function to traverse correct N'-paths during the computation of WCC which is explained in the next section.

Alg. 2 is a worklist-based algorithm that visits each edge (n, m) in the CFG exactly once, detects and includes an edge (m, n', l) in the influencer graph $\mathcal{G}$ for any $l \in \{0, 1, 2, 3\}$ and $n' \in N$ that may influence m where $n' = n$ or $n' = \mathcal{I}(n)$ (see Lines 13-14, 21, 27, 31, 36). It also computes the minimum distance $\mathcal{D}ist(m, n')$ from n' to m or a Cond node immediately before m in the

Input: CFG $G = (N, E)$
Output: $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $\mathcal{D}ist$

forall $(n \in N$ **do**
  $\mathcal{I}(n) = \perp$, visit(n)=false
end
$W = \{(n_{\triangleright}, 0, 0)\}$ and $\mathcal{I}(n_{\triangleright}) = \iota$
while $(W \neq \emptyset)$ **do**
  Remove (n, l, d) from W
  visit(n)=true
  $newlabel = 1$
  forall $(m \in succ(n))$ **do**
    $nedge = (\perp, \perp)$
    if (n *is a Cond node*) **then**
      $\mathcal{I}(m) = n$
      if $((n, X) \notin \mathcal{E}_m$ *for any* $X \in \{0, 1, 2, 3\}$ **then**
        $nedge = (n, newlabel)$
      if $((n, l') \in \mathcal{E}_m \wedge l' > 0$ *such that* $l' \neq newlabel)$ **then**
        $nedge = (n, 3)$
      $newlabel = newlabel + 1$
      $dist = 1$
    end
    if (n *is a nonCond Join node*) **then**
      $\mathcal{I}(n) = n$
      $\mathcal{I}(m) = n$
      $nedge = (n, 0)$
      $dist = d + 1$
    end
    if (n *is not a Cond and Join node*) **then**
      $\mathcal{I}(m) = \mathcal{I}(n)$
      if $(\mathcal{I}(n) \neq \iota \wedge \neg(\mathcal{I}(n), X) \in \mathcal{E}_m$ *for any* $X \in \{0, 1, 2, 3\})$ **then**
        $nedge = (\mathcal{I}(n), l)$
        $dist = d + 1$
      end
      if $(\mathcal{I}(n) \neq \iota \wedge (\mathcal{I}(n), l') \in \mathcal{E}_m \wedge l' > 0 \wedge l' \neq l)$ **then**
        $nedge = (\mathcal{I}(n), 3)$
        $dist = min(d + 1, \mathcal{D}ist(m, \mathcal{I}(n))$
      end
    end
    if $(nedge = (m', l') \neq (\perp, \perp))$ **then**
      $\mathcal{E}_m = \mathcal{E}_m \cup \{nedge\}$
      $\mathcal{D}ist(m, m') = dist$
    end
    if $(\neg visit(m) \wedge nedge = (m', l'))$ **then** $W = W \cup \{(m, l', dist)\}$
    else if $(\neg visit(m)$ **then** $W = W \cup \{(m, l, dist)\}$
  end
end

Algorithm 2: genInfluencerGraph

CFG. The choice of n' depends on the kind of visited node n. If n is a Cond node or a Join node, then $n' = n$ as n is recognized to be the new influencer of m, and $n' = \mathcal{I}(n)$ otherwise to interpret the fact that the influencer of n becomes the influencer of m. The choice of l depends on the following facts:

- If n is a Cond node, then the first and the second visited successors m_1 and m_2 of n introduce the edges $(m_1, n, 1)$ and $(m_2, n, 2)$ in $\mathcal{G}$. However, we include the edge $(m_i, n, 3)$ instead of (m_i, n, l) for $l = 1$ or $l = 2$ if there already exists an edge (m_i, n, l') in $\mathcal{G}$ for any $i = 1, 2$, $l' > 0$, and $l' \neq l$ since m_i can be reached from n through either of the two branches.
- If n is a Join node and not a Cond node, we include the edge $(m, n, 0)$ in $\mathcal{G}$ to represent the fact that m is not influenced by n, but there possibly be a successor of n in $\mathcal{G}$ that may influence m.
- If n is neither a Join node nor a Cond node, then we include the edge $(m, \mathcal{I}(n), l)$ in $\mathcal{G}$ where the edge-label l may be the continuation of the previously selected edge-label or $l = 3$ if $\mathcal{G}$ already includes an edge $(m, \mathcal{I}(n), l')$ for any $l' > 0$ and $l' \neq l$.

3.2 An incremental algorithm to compute WCC

Once we obtain the influencer graph $\mathcal{G}$ as explained in the previous section, we perform a search for disjoint N'-paths in $\mathcal{G}$ to detect all N'-weakly deciding nodes. Then, the WCC of N' includes N' and all N'-weakly deciding nodes that are reachable from N' in the CFG. Before we provide a systematic search in graph $\mathcal{G}$, we provide a few details on $\mathcal{G}$.

There may have multiple edges (n_1, n, l) and (n_2, n, l) to the node n in $\mathcal{G}$ with the same edge label. This is due to the sequence of CFG nodes n, n_1, n_2 in the CFG such that n is a Cond or Join node, and n_i is neither a Cond nor a Join node for $i = 1, 2$. Thus, both n_1 and n_2 belong to the same branch of n and both are predecessor nodes of n in $\mathcal{G}$. If either n_1 or n_2 belongs to N', then the edge (n_i, n, l) represents the N'-path $[n..n_i]$ for any $i \in \{1, 2\}$. However, if both n_1 and n_2 are in N', then only $[n..n_1]$ is an N'-path, but $[n..n_2]$ is not. Our systematic search in $\mathcal{G}$ incorporates this fact by looking into the distance $\mathcal{D}ist(n_i, n)$ computed in Alg. 2 and considers the N'-path $[n..n_1]$ as $\mathcal{D}ist(n_1, n) < \mathcal{D}ist(n_2, n)$.

We compute the set of N'-weakly deciding vertices by traversing the graph $\mathcal{G}$ from the nodes in N' in the forward direction according to Alg. 4. We maintain the following functions as invariant during the fixpoint iteration of the algorithm:

- The function $\mathcal{E}nd : N \to N$ records the end element of a potential N'-path from the given node. For example, $\mathcal{E}nd(n) = m$ implies that $[m..n]$ is a potential N'-path in the CFG. Initially, we set $\mathcal{E}nd(n) = \perp$ for each CFG node n to represent the fact that an N'-path from n is yet to be traversed (if exists).
- For all CFG nodes n such that either $n \in N'$ or n is identified as an N'-weakly deciding node, we set $wdVec(n) = true$; $wdVec(n) = false$ otherwise.

- The function $\mathcal{L} : N \to \mathcal{P}(\{1, 2, 3\})$ records the set of non-zero edge-labels of a given node n, and any $l \in \mathcal{L}(n)$ indicates that an N'-path exists from n in the branch of the CFG marked by l. For example, $\mathcal{L}(n) = \{1, 3\}$ indicates that an N'-path exists from node n in the CFG which can be visited by traversing any of the two branches of n in the CFG.
- The function $\mathcal{M} : N \times \{0, 1, 2, 3\} \to N \times \mathbb{N}$ represents the pair of a CFG node and a nonnegative integer number in relation to a given pair of a CFG node and an edge label. Given any CFG node m and the edge label $l \in \{0, 1, 2, 3\}$, $\mathcal{M}(m, l) = (n, d)$ represents the fact that (n, m, l) is an edge in the influencer graph $\mathcal{G}$ with distance $d = \mathcal{D}ist(n, m)$ such that the path $[m..n]$ is either an N'-path or a prefix of an N'-path. If there exist multiple edges (n', m, l) and (n, m, l) in $\mathcal{G}$ such that both represent N'-paths (or a prefix of N'-paths) $[m..n]$ and $[m..n']$, we set $\mathcal{M}(m, l) = (n, d)$ if $d = \mathcal{D}ist(n, m) < \mathcal{D}ist(n', m)$. We set $\mathcal{M}(m, l) = (\perp, \perp)$ when no N'-path exists from m in the CFG. If $\mathcal{M}(m, l) = (n, d)$, we use the functions $first$ and $second$ to denote the equality $first(\mathcal{M}(m, l)) = n$ and $second(\mathcal{M}(m, l)) = d$ respectively.

Given the CFG G, a subset of CFG nodes N', and the boolean variable *initialize?*, Alg. 3 computes the WCC of N'. If *initialize?* is *true*, the functions $\mathcal{E}nd, \mathcal{L}, wdVec$, and $\mathcal{M}$ are initialized, and the influencer graph $\mathcal{G}$ and the function $\mathcal{D}ist$ are computed by applying Alg. 2. Alg. 4 is applied to compute the set of N'-weakly deciding nodes WD followed by computing the reachability of the nodes in WD from N' in the CFG G. We omit the details of the CHECKREACHABILITY function as it is a simple graph reachability algorithm visiting each edge in the CFG exactly once starting from N' and return the set of all nodes in WD that are reachable from N'.

For the subsequent application of Alg. 3 in computing the WCC of a superset of N' (after the first computation of WCC set), Alg. 3 is called with the boolean variable *initialize?* set to *false* . Alg. 4 is applied with the previously computed values of the functions $\mathcal{E}nd, \mathcal{L}, wdVec$, and $\mathcal{M}$. These functions are considered as the internal states of the algorithm which are initialized only once and the WCC set is computed incrementally if the input set N' grows incrementally in the consecutive calls of Alg. 3.

Alg. 4 computes the set of N'-weakly deciding nodes from the influencer graph $\mathcal{G}$ and the set N'. We assume that the functions $\mathcal{E}nd, \mathcal{L}, wdVec, \mathcal{M}$, and $\mathcal{D}ist$ are globally available to Algorithms 3 - 5. Alg. 4 systematically traverses the graph $\mathcal{G}$ from the nodes in N' in the forward direction and updates the functions $\mathcal{E}nd, \mathcal{L}, wdVec, \mathcal{M}$ to record the visit of N'-paths and their disjointedness. While visiting an edge (n, m, l) in $\mathcal{G}$, $\mathcal{L}(m)$ is updated to record the visit of the node m through the branch $l \neq 0$, $\mathcal{M}(m, l)$ is updated to record the predecessor node n of m providing the N'-path through the branch l and the distance from n (or the nearest Cond node of n) to m. A CFG node m is included in the set of N'-weakly deciding vertices WD if the following constraints are satisfied:

$$|\mathcal{L}(m)| > 1 \tag{2}$$

$$|\{\mathcal{E}nd(p) : l' \in \mathcal{L}(m), p = first(\mathcal{M}(m, l')), \mathcal{E}nd(p) \neq \perp\}| > 1 \tag{3}$$

Input: $G, N', initialize?$
Output: WCC
if $(initialize?)$ **then**
  forall $(n \in N)$ **do**
    $\mathcal{E}nd(n) = \perp$, $\mathcal{L}(n) = \emptyset$, $wdVec(n) = false$
    forall $l \in \{0, \ldots, 3\}$ **do** $\mathcal{M}(n, l) = (\perp, \perp)$
  end
  $(\mathcal{G}, \mathcal{D}ist) =$GENINFLUENCERGRAPH$(G)$
end
$WD =$COMPUTEWD$(\mathcal{G}, N')$
$WCC = N' \cup$ CHECKREACHABILITY(G, WD, N')

Algorithm 3: COMPUTEWCC

Input: $\mathcal{G} = (N, \mathcal{E}), N'$
Output: WD
$WD = \emptyset$
forall $(n \in N')$ **do**
  $wdVec(n) = true$, $\mathcal{E}nd(n) = n$
end
$W = N'$
while $(W \neq \emptyset)$ **do**
  Remove n from W
  forall $((m, l) \in \mathcal{E}_n)$ **do**
    if $(wdVec(m))$ **then** continue
    if $(l \neq 0)$ **then** $\mathcal{L}(m) = \mathcal{L}(m) \cup \{l\}$
    if $(\mathcal{M}(m, l) = (\perp, \perp) \vee second(\mathcal{M}(m, l)) > \mathcal{D}ist(n, m))$ **then**
      $\mathcal{M}(m, l) = (n, \mathcal{D}ist(n, m))$
    end
    $changed = false$
    $S = \{\mathcal{E}nd(p) : l' \in \mathcal{L}(m), p = first(\mathcal{M}(m, l')), \mathcal{E}nd(p) \neq \perp\}$
    if $(|S| > 1)$ **then**
      $errorEnd = \mathcal{E}nd(m)$
      $\mathcal{E}nd(m) = m$
      $WD = WD \cup \{m\}$
      $wdVec(m) = true$
      $changed = true$
      if $(errorEnd \neq \perp \wedge errorEnd \neq m)$ **then**
        propagateToReplace$(m, errorEnd)$
      end
    end
    else
      $changed = (\mathcal{E}nd(m)! = \mathcal{E}nd(first(\mathcal{M}(m, l)))$
      $\mathcal{E}nd(m) = \mathcal{E}nd(first(\mathcal{M}(m, l)))$
    end
    if $(changed)$ **then** $W = W \cup \{m\}$
  end
end

Algorithm 4: COMPUTEWD

Input: $m, errorEnd$
1 $W_q = \{m\}$
2 **while** $(W_q \neq \emptyset)$ **do**
3 Remove n from W_q
4 **for** $(m \in succ_{\mathcal{G}}(n))$ **do**
5 **if** $(\mathcal{E}nd(m) == errorEnd \wedge wdVec(m) == false)$ **then**
6 $\mathcal{E}nd(m) = \perp$
7 $W_q = W_q \cup \{m\}$;
8 **end**
9 **end**
10 **end**

Algorithm 5: propagateToReplace

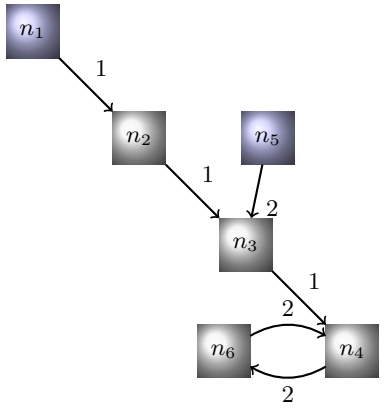

Fig. 3: An example influencer graph

Equation 2 implies that m is a Cond node in the CFG having two branches and multiple N'-paths exit from m in the CFG. Equation 3 implies that there exist two predecessors m_1 and m_2 of m in $\mathcal{G}$ such that $\mathcal{E}nd(m_1) \neq \mathcal{E}nd(m_2)$ and $\mathcal{E}nd(m_i) \neq \perp$ for $i = 1, 2$. Thus, there exist two disjoint N'-paths from m in the CFG, and m is included in the set of an N'-weakly deciding vertices WD. We provide the theoretical proof of the correctness of our approach in the next section.

While visiting the graph $\mathcal{G}$, it may happen that the update of the function $\mathcal{E}nd$ is based on partial information that may lead to error if proper actions are not taken. We illustrate this scenario by an example in Fig. 3. Let n_1 and n_5 belong to N' and $\mathcal{E}nd(n_i) = \perp$ for all $1 \leq i \leq 6$ initially. During the visit of this graph from n_5, Alg. 4 updates $\mathcal{E}nd(n_i) = n_5$ for all $i \in \{3, \ldots, 6\}$. While visiting the graph from n_1, eventually node n_3 is included in WD due to visiting two disjoint paths $[n_3..n_5]$ and $[n_3..n_1]$, and $\mathcal{E}nd$ is updated by $\mathcal{E}nd(n_3) = n_3$. Next, if node n_4 is visited without taking any action, Equations 2 and 3 are satisfied for node n_4 since $\mathcal{E}nd(n_6) = n_5$ and $\mathcal{E}nd(n_3) = n_3$, and node n_4 will be included

in *WD* imprecisely. Even though two N'-paths exist from n_4, the paths are not disjoint. We resolve this imprecision in our algorithm as follows. If a CFG node m is included in *WD*, we set $\mathcal{E}nd(m) = m$ due to our choice of invariant during the fixpoint computation. Moreover, node m should be the end element for all nodes $n \notin WD$ and that belong to a path π from m in $\mathcal{G}$. However, if we had $\mathcal{E}nd(m) = m' \neq \bot$ before we set $\mathcal{E}nd(m) = m$, and if we have $\mathcal{E}nd(n) = m'$, we reset $\mathcal{E}nd(n) = \bot$ so that $\mathcal{E}nd(n)$ can be set to m in a later visit to n from m. For example, for the influencer graph in Fig. 3, we reset $\mathcal{E}nd(n) = \bot$ and later set $\mathcal{E}nd(n) = n_3$ for $n \in \{n_4, n_6\}$ after we set $\mathcal{E}nd(n_3) = n_3$. This action will not include n_4 in *WD* as Equation 3 will not be satisfied for node n_4 due to resetting $\mathcal{E}nd(n_6)$. Alg. 4 performs this reset by calling the *propagateToReplace* procedure in Alg. 5.

3.3 Proof of correctness

In this section, we provide a number of lemmas to justify that Alg. 4 correctly discovers all N'-weakly deciding nodes. Finally, we provide Theorem 1 to prove that Alg. 3 correctly computes the WCC of N'.

Lemma 1. *Any path $[n_1..n_k]$ in the influencer graph $\mathcal{G}$ implies that there exists a path $[n_k..n_1]$ in the CFG.*

Proof. Let $n \in N$ be a CFG node such that $n_i \in succ(n)$ for any $1 \leq i \leq k-1$. Alg. 2 includes the edge (n_i, n_{i+1}) in $\mathcal{G}$ while visiting the CFG node n due to one of the following cases:

1. If n is a Cond node or a Join node, then $n_{i+1} = n$ and n_{i+1}, n_i is a path in the CFG,
2. Otherwise, $n_{i+1} = \mathcal{I}(n)$.

Let $\mathcal{I}(n) = m_1$ for any $m_1 \in N$. According to Alg. 2, m_1 can only be a Join or a Cond node (see Lines 12, 19-20). Then, $\mathcal{I}(n) = m_1$ is obtained due to traversing a CFG path $m_1, \ldots, m_l = n$ such that no m_j is a Cond or Join node and $\mathcal{I}(m_j) = m_1$ (Line 25) for $2 \leq j \leq l$. Thus, there exists CFG paths $[m_1..n]$, and consequently, $[m_1..n_i]$ is a CFG path with $n_{i+1} = m_1$ being a Cond node. Thus, any path $[n_1..n_k]$ in the influencer graph $\mathcal{G}$ is obtained due to the existence of a CFG path $[n_k..n_1]$ such that $n_2, \ldots, n_k$ are all Cond or Join nodes. □

Note. *While constructing the graph $\mathcal{G}$, an edge (n,m) is included in this graph if either m is a Cond or Join node, or $m = I(n')$ for some node n' such that $I(n') \neq \bot$. If $I(n') = \bot$, we would not have an edge (n,m) in this graph (see the conditions at lines 26, 30, and 35 in Alg. 2.*

Lemma 2. *Let $N' \subseteq N$, let $[n_1..n_k]$ be any path in the influencer graph $\mathcal{G}$, let $n_1 \in N'$, and let $n_i \notin N'$ for all $2 \leq i \leq k$. Then, either $\pi = [n_k..n_1]$ is an N'-path or there exists another N'-path which is a prefix of π in the CFG G.*

Proof. According to Lemma 1, there exists a path $[n_k..n_1]$ in the CFG G. If no node in this path is in N' except n_1, then $[n_k..n_1]$ is an N' path. However, if the path $[n_k..n_1]$ includes multiple nodes from N', then there exists a node $m \in [n_k..n_1]$ which is in N' and the closest node to n_k. Thus, $[n_k..m]$ is an N'-path in the CFG which is the prefix of $[n_k..n_1]$. □

Lemma 3. *Let $\mathcal{E}nd(n) = m$ for any $n, m \in N$ computed in Alg. 4. Then, there exists a path $\pi = [m..n]$ in the influencer graph $\mathcal{G}$ such that either $m \in N'$ or $m \in WD$.*

Proof. Alg. 4 assigns $\mathcal{E}nd(n) = m$ while traversing an edge (n', n, l) in $\mathcal{G}$ for any $n' \in N$. Either (i) $\mathcal{E}nd(n) = n$ (Line 18) or (ii) $\mathcal{E}nd(n) = \mathcal{E}nd(n_1) = m$ for any predecessor $n_1 = first(\mathcal{M}(n, l)$ of n (Line 28). In the first case, $n = m$ and $n \in WD$. Thus, π is a trivial path containing only the node n, and the lemma trivially holds. In the second case, we use the inductive reasoning to show that there exists a path $n_k, \ldots, n_1$ in $\mathcal{G}$ such that $\mathcal{E}nd(n_i) = \mathcal{E}nd(n_{i+1})$ for all $1 \leq i \leq k-1$ due to the update in Line 28, and eventually we must have $n_k = m$ and $\mathcal{E}nd(m) = m$ since $\mathcal{G}$ is a finite graph. Alg. 4 assigns $\mathcal{E}nd(m) = m$ if $m \in N'$ (Line 3) or $m \in WD$ (Line 18) during traversing $\mathcal{G}$, and consequently, the lemma holds. □

Lemma 4. *Let $\mathcal{E}nd(n) \neq \perp$ for any $n \in N$ computed in Alg. 4. Then, there exists an N'-path from n in the CFG.*

Proof. Let $n = n_0$, and let $\mathcal{E}nd(n_0) = n_1$ for any $n_1 \in N$. According to Lemma 3, there exists a path $[n_1..n_0]$ in the influencer graph $\mathcal{G}$ such that either $n_1 \in N'$ or $n_1 \in WD$. If $n_1 \in N'$, then according to Lemma 2, there exists an N'-path from n in the CFG G. However, if $n_1 \in WD$, there exists a predecessor m_1 of n_1 in $\mathcal{G}$ such that $\mathcal{E}nd(m_1) \neq \perp$. Let $\mathcal{E}nd(m_1) = n_2$. We apply the inductive reasoning to infer that there exists a sequence of paths $[n_k..n_{k-1}], \ldots, [n_1..n_0]$ in $\mathcal{G}$ such that $n_i \in WD$ for $0 \leq i \leq k-1$, and eventually $n_k \in N'$ since $\mathcal{G}$ is finite, $\mathcal{E}nd(m) = m$ for all $m \in N'$ due to initialization (Line 3 in Alg. 4), and $\mathcal{E}nd(m')$ for all $m' \notin N'$ are updated from $\mathcal{E}nd(m)$ by traversing $\mathcal{G}$ from N'. Thus, according to Lemma 2, there exists an N'-path from n in the CFG. □

Lemma 5. *Let $n \in N$ be a CFG node satisfying Equation 2 and Equation 3. Then, there exist two edges (n_1, n, l_1) and (n_2, n, l_2) in $\mathcal{G}$ such that $l_1 \neq l_2$, $\mathcal{E}nd(n_1) \neq \mathcal{E}nd(n_2)$, $\mathcal{E}nd(n_i) \neq \perp$ and $l_i > 0$ for $i = 1, 2$.*

Proof. Since $|\mathcal{L}(n)| > 1$, there exist two edges (n_1, n, l_1) and (n_2, n, l_2) in $\mathcal{G}$ for any $n_1, n_2 \in N$ such that $l_1 \neq l_2$ and $l_i > 0$ for $i = 1, 2$. The conditions $l_1 \neq l_2$ and $l_i > 0$ for $i = 1, 2$ imply that node n is a Cond node having two successors in the CFG.

Since Equation 3 is satisfied for the node n, either $\mathcal{E}nd(n_1) \neq \mathcal{E}nd(n_2)$ and the lemma holds consequently, or $\mathcal{E}nd(n_1) = \mathcal{E}nd(n_2)$. In the second case, there exists another edge (n_3, n, l_3) such that $\mathcal{E}nd(n_3) \neq \mathcal{E}nd(n_i)$ for $i = 1, 2$. As n is a Cond node, $l_3 \neq 0$ according to Alg. 2 (see Lines 13 and 14 in Alg. 2). So, we have one of the following possibilities: (i) $l_3 \neq l_1$ and $l_3 \neq l_2$, (ii) $l_3 = l_1$, but

$l_3 \neq l_2$, or (iii) $l_3 = l_2$, but $l_3 \neq l_1$. Consequently, we have the CFG nodes n_3 and either n_2 due to Case (i) or (ii) or n_1 due to Case (iii) such that the conditions in the lemma are satisfied. □

Lemma 6. *Let $\pi = [n..m]$ be a CFG path such that n is reachable from the Start node. Then, there exists a subsequence of CFG nodes $n_1, \ldots, n_k = m$ of π such that $[n_k..n_1]$ is a path in $\mathcal{G}$.*

Proof. We consider the subsequence of CFG nodes $n_1, \ldots, n_k = m$ of π such that each n_i is a Cond or Join node for $1 \leq i \leq k-1$. Let $m_0^i = n_i, \ldots, m_{i_k}^i = n_{i+1}$ be the sequence of nodes from n_i to n_{i+1} in π for all $1 \leq i \leq k-1$ and $i_k \geq 0$. Alg. 2 traverses the path π as it is reachable from the Start node. At each visit to the node m_j^i for any $1 \leq j \leq i_k$, Alg. 2 inserts the edge (m_j^i, n_i, l) for any $l \in \{0, \ldots, 3\}$. Thus, (n_{i+1}, n_i, l) is an edge in $\mathcal{G}$ for all $1 \leq i \leq k-1$, and $[n_k..n_1]$ is a path in $\mathcal{G}$. □

Lemma 7. *Let $n \in N$ be a CFG node such that there exist two disjoint N'-paths from n in the CFG. Then, Equations 2 and 3 are satisfied for n in Alg. 4.*

Proof. Let π_1 and π_2 be two disjoint N'-paths from n in the CFG. According to Lemma 6, there exist two paths $[n_k..n_1]$ and $[m_l..m_1]$ in $\mathcal{G}$ which are subsequences of nodes in π_1 and π_2 respectively. Also, $n_k, m_l \in N'$ since π_1 and π_2 are N'-paths in the CFG.

Node n is a Cond node having two distinct branches in the CFG, and thus we must have $n_1 = n$ and $m_1 = n$. So, there exist two edges (n_2, n, l_1) and (m_2, n, l_2) in the influencer graph $\mathcal{G}$ such that $l_1 \neq l_2$ since n_2 and m_2 are at different branches of n. Also, since n is a Cond node, $l_i > 0$ for $i = 1, 2$. Thus, Equation 2 is satisfied for n.

Since π_1 and π_2 are disjoint paths that only meet at the CFG node n, $[n_k..n_1]$ and $[m_l..m_1]$ are also disjoint paths that only meet at $n_1 = m_1$. We must have $\mathcal{E}nd(n_k) = n_k$ and $\mathcal{E}nd(m_l) = m_l$ since $n_k, m_l \in N'$. While visiting the influencer graph $\mathcal{G}$ in Alg. 4, $\mathcal{E}nd(n_2)$ and $\mathcal{E}nd(m_2)$ may take any value from the sets $\{n_2, \ldots, n_k\}$ and $\{m_2, \ldots, m_l\}$ respectively, which are disjoint sets. Thus, Equation 3 is satisfied for n. □

Lemma 8. *Let $n \in N$ be a CFG node satisfying Equation 2 and Equation 3. Then, n is an N'-weakly deciding node.*

Proof. According to Lemma 5, there exist two edges (n_1, n, l_1) and (m_1, n, l_2) in $\mathcal{G}$ such that $l_1 \neq l_2$, $\mathcal{E}nd(n_1) \neq \mathcal{E}nd(m_1)$, $\mathcal{E}nd(n_1) \neq \bot, \mathcal{E}nd(m_1) \neq \bot$ and $l_i > 0$ for $i = 1, 2$.

For any $m \in N'$, $\mathcal{E}nd(m) = m$ due to initialization, and $\mathcal{E}nd(m') \neq \bot$ is derived from this initial values of $\mathcal{E}nd(m)$ for all other $m' \notin N'$. Let $\mathcal{E}nd(n_1) = n_2$. There exists a path $[n_2..n_1]$ in $\mathcal{G}$ such that either $n_2 \in N'$ or $n_2 \in WD$ (Lemma 3). If $n_2 \in WD$, there exist a predecessor n_2' of n_2 such that $\mathcal{E}nd(n_2') \neq \bot$ according to Alg. 4 (see Lines 15-18 in Alg. 4). Let $\mathcal{E}nd(n_2') = n_3$. Since $\mathcal{G}$ is finite, we apply inductive reasoning to infer that there exists a path $[n_k..n_1]$ in $\mathcal{G}$ such that $n_k \in N'$, $[n_{i+1}..n_i]$ is a path in $\mathcal{G}$ and $n_i \in WD$ for all $1 \leq i \leq k-1$

. Similarly, there exists a path $[m_l..m_1]$ in $\mathcal{G}$ such that $m_l \in N'$, $[m_{i+1}..m_i]$ is a path in $\mathcal{G}$ and $m_i \in WD$ for all $1 \leq i \leq l-1$.

In what follows, we show that the paths $\pi_1 = [m_l..m_1, n]$ and $\pi_2 = [n_k..n_1, n]$ are disjoint by contradiction. Suppose there exist no such disjoint paths to n. Thus, they meet at the first common node $n_i = m_j$ in π_1 and π_2 for any $1 \leq i \leq k$ and $1 \leq j \leq l$. So, the paths $[n_k..n_{i+1}]$ and $[m_l..m_{j+1}]$ are disjoint.

Now, if there exists a node $m \in [n_i..n]$ which is in WD and thus $\mathcal{E}nd(m) \neq \perp$, we can apply inductive reasoning as before to show that there exists a path $[m^p..m^1]$ for any $p \geq 1$ such that $m_p \in N'$. This path does not meet any of the paths $[n_k..n_{i+1}]$ and $[m_l..m_{j+1}]$ as they are disjoint. This implies that we can always have two disjoint paths from n regardless of whether $[m^p..m^1]$ meet with $[n_i..n]$ and/or $[m_j..n]$:

1. the path $[n_k..n]$ if $[m^p..m^1]$ meet with both $[n_i..n]$ and $[m_j..n]$ or only with $[m_j..n]$, and
2. another path $[m^p..m^t]$ followed by $[m^t..n]$ where m^t meet at $[m_j..n]$.

This contradicts our assumption that no paths like π_1 and π_2 to n are disjoint. So, our assumption about the existence of the node $m \in WD$ is not correct. Thus, no node in the paths $[n_i..n]$ and $[m_j..n]$ are in WD. We must have $\mathcal{E}nd(n_i) \neq \perp$ due to the path $[n_k..n_i]$ such that $\mathcal{E}nd(n_k) = n_k$. Alg. 4 then traverses the paths $[n_i..n]$ and $[m_j..n]$ and set $\mathcal{E}nd(m) = \mathcal{E}nd(n_i)$ for all $m \in [n_i..n]$ and $m \in [m_j..n]$ (see conditions at Line 16 and the update at Line 28 in Alg. 4). This contradicts the assumption of the lemma that $\mathcal{E}nd(n_1) \neq \mathcal{E}nd(n_2)$. Thus, our only assumption that π_1 and π_2 are not disjoint cannot be true.

Then, π_1 and π_2 lead to two N'-paths π_3 and π_4 from n in the CFG G according to Lemma 4. These paths are disjoint as all Cond and Join nodes in these paths are disjoint. Node n is thus an N'-weakly deciding vertex. □

Theorem 1. *Alg. 3 correctly and precisely computes the WCC of a subset N' of CFG nodes.*

Proof. For all N'-weakly deciding node n in the CFG, Equations 2 and 3 are satisfied for n in Alg. 4 according to Lemma 7. Moreover, for all CFG nodes n satisfying Equations 2 and 3 in Alg. 4, n is an N'-weakly deciding node according to Lemma 8. Since Alg. 4 includes all CFG nodes n in WD if Equations 2 and 3 are satisfied for n, Alg. 4 includes a CFG node n in WD if and only if n is an N'-weakly deciding node. Alg. 3 then computes the set WCC that includes N' and a subset of WD that are reachable from N' in the CFG. □

3.4 Worst-case time complexity

In this section, we provide a number of lemmas to state the theoretical worst-case time complexity of our algorithms.

Lemma 9. *Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be the influencer graph generated from a CFG $G = (N, E)$ such that $|\mathcal{G}| = |\mathcal{N}| + |\mathcal{E}|$ and $|G| = |N| + |E|$. Then, it holds that $O(|\mathcal{G}|) = O(|G|)$.*

Proof. For each visit to a CFG edge, Alg. 2 inserts at most one edge in $\mathcal{G}$. Moreover, since $N = \mathcal{N}$, we must have $O(|\mathcal{G}|) = O(|G|)$. □

Lemma 10. *The worst-case time complexity of Alg. 2 is* $O(|N|)$.

Proof. The worst-case time complexity of Alg. 2 is dominated by the ***while*** loop (line 5-42). The ***while*** loop visits each CFG node exactly once. For each visited node n, the ***forall*** loop (line 9-41) visits each edge from n exactly once. Thus, the ***while*** and the ***forall*** loop collectively iterates $|N| + |E|$ times. The worst-case cost of each basic operation inside the ***while*** and the ***forall*** loop is constant except for the costs of the operations $(n, X) \in \mathcal{E}_m$ and $(\mathcal{I}(n), X) \in \mathcal{E}_m$ at Lines 13, 14, 26, and 30, the cost of inserting the edge $\{nedge\}$ in $\mathcal{E}_m$ at Line 36, and the cost of accessing and updating $\mathcal{D}ist(m, .)$ at Lines 32 and 37. Since each CFG node and each edge in the CFG is visited at most once, the cost of these operations during the entire iteration of the ***while*** and the ***forall*** loop is $O(|\mathcal{G}|)$ which is equal to $O(|G|)$ according to Lemma 9. Thus, the worst-case cost of the entire loop (Line 5-42) is $O(|N| + |E| + |G|)$ which is equal to $O(|N| + |E|)$. Since any CFG node has at most two outgoing edges according to the definition of CFG, we have $O(|N|) = O(|E|)$, and the worst-case time complexity of Alg. 2 is $O(|N|)$. □

Lemma 11. *The worst-case time complexity of Alg. 4 is* $O(|N|^2)$.

Proof. The worst-case cost of Alg. 4 is dominated by the ***while*** loop (Line 6-32). This loop iterates as long as there exist elements in the worklist W. While visiting an edge (n, m, l) in the influencer graph $\mathcal{G}$, m is included in W if $\mathcal{E}nd(m)$ is changed (see Lines 18, 21, 27-30). The function $\mathcal{E}nd(m)$ may change its value $\perp$ to some other value $n \in N$ for the first time. If $\mathcal{E}nd(m) = m$ is set once, its value will never be changed and it will never be included in W. Also, $\mathcal{E}nd(m) = n$ may change to $\mathcal{E}nd(m) = n'$ if there exists a path $[n..m]$ in the influencer graph such that $n' \in [n..m]$ and n' is included in WD, or $n' \in N'$ is an immediate predecessor of m. Thus, if $\mathcal{E}nd(m)$ is changed $|N|$ times, it implies that $|N|$ nodes are in $WD \cup N'$ and thus no new node can be included in W as $\mathcal{E}nd$ will then never be changed, and the *changed* variable will always be false afterward. So, we can safely consider that for each node m in the influencer graph, $\mathcal{E}nd(m)$ may change two times (from $\perp$ to some value $n \in N$, and n to m if $m \in WD$). Each additional change will be due to including a node in WD. Since at most $|N|$ node can be included in WD, the ***while*** and the loop ***forall*** loop (Lines 8-31) will iterate at most $2 * |N| + 2 * |\mathcal{E}|$ times in total.

By choosing suitable data structures, all other operations in the ***while*** and ***forall*** loops can be performed at constant time except for the operation of the *propagateToReplace* procedure in Alg. 5. Alg. 5 requires visiting each node and edge in the graph $\mathcal{G}$ at most once with all other operations in the ***while*** loop at constant cost. Thus, the worst-case cost of Alg. 5 is $O(|\mathcal{N}| + |\mathcal{E}|)$ which is effectively $O(|N|)$ due to Lemma 9. Thus, the worst-case cost of the ***while*** loop in Alg. 4 is $O((2 * |N| + 2 * |\mathcal{E}|) * |N|)$ which is equivalent to $O(|N|^2)$. This is also the worst-case cost of Alg. 4. □

Theorem 2. *The worst-case time complexity of Alg. 3 computing the WCC set is* $O(|N|^2)$.

Proof. The worst-case cost of the CHECKREACHABILITY procedure can be at most $O(|N|)$ times as it requires visiting each node and edge at most once in a loop with all other operations in the loop at a constant cost. Thus, the worst-case cost of Alg. 3 is dominated by the worst-case cost of the COMPUTEWD procedure which is $O(|N|^2)$ according to Lemma 11.

Note that even though the worst-case cost of Alg. 3 is quadratic in the size of the CFG, we believe that the amortized complexity of this algorithm is much better as indicated by our experimental evaluation in the next section.

4 Experimental Evaluation

The main objectives of our experimental evaluation include measuring the correctness of our algorithm and comparing its practical efficiency with the state-of-the-art approaches. In doing so, we have implemented our algorithms in the Clang/LLVM compiler framework [9]. We have compared our approach with the state-of-the-art WCC computation algorithm developed earlier [14,12] which is currently the best-known algorithm for computing WCC with an average speedup of 10.6 compared to the algorithm of Danicic et al. [4]. This state-of-the-art algorithm is also implemented in the Clang/LLVM compiler framework and released as open-source in a GitHub repository[2].

All experiments are performed in an Intel(R) Core(TM) i7-7567U 3.50GHz CPU with 16 GB of RAM memory and all implementations are compiled using the LLVM version 11.0.0. We have used seven benchmarks from the SPEC CPU 2017 benchmark suite consisting of approximately 2081 KLOC. These benchmarks are written in C language and were also used in the experimental evaluation of the state-of-the-art approaches. Note that the SPEC CPU 2017 benchmark suite contains other benchmarks. However, they are not written solely in the C language. Since our implementation can only handle C code, these other benchmarks are thus excluded from the experiments.

In order to perform experiments for the incremental computation of WCC, we choose the set N' of CFG nodes randomly. This choice of randomness is due to the fact that N' should be provided by the client application of WCC such as program slicing as illustrated in Alg. 1. This choice of randomness neither affects the generality of our algorithm nor affects our experiment in any way. We run each experiment 10 times. The number 10 is selected due to the fact that earlier experimental evaluation [14,12] ran each experiment 10 times as well. For the client applications like slicing or information flow control, this number will depend on the size of the CFG, the maximal number of Cond nodes in a maximal path, and the point of interest such as the nodes in the slicing criterion, etc. Usually, this number should be the maximum number of iterations to reach

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

#	benchmarks	KLOC	#proc	T_ω	T_M	speedup
1	Mcf	3	40	17728.4	52010.9	2.93
2	Nab	24	327	121747.7	430778.8	3.54
3	Xz	33	465	66832.0	147705.6	2.21
4	X264	96	1449	60603.5	208273.6	3.44
5	Imagick	259	2586	56359.8	225375.0	4.0
6	Perlbench	362	2460	2006511.4	18762418.0	9.35
7	GCC	1304	17827	12317538.2	434684910.3	35.29
	Total/Average	2081	25154	14647321.0	454511472.2	31.03

Table 1: Execution times of computing WCC incrementally on seven selected benchmarks from SPEC CPU 2017 [3]

fixed-point in Alg. 1. However, since this number is unknown, 10 is a good number for the experiments to get an indication of whether we can obtain a significant speedup or not. Each experiment took the Z number of randomly selected N' sets where Z is a random number between 1 and 15.

We computed the influencer graph only once and apply our WCC computation algorithm (Alg. 3) Z times consecutively for each experiment. On the other hand, we apply the state-of-the-art algorithm [14,12] Z times consecutively to compute the WCC set for each experiment. In order to verify the correctness of our method, we compare the WCC sets computed by our incremental algorithm and the best baseline algorithm in [14,12] computed for the same N' set. We obtained exactly the same WCC sets for all experiments in each benchmark computed by both methods. This provides us the empirical proof of the correctness of our method.

For each experiment, we recorded the time in milliseconds taken by both methods. The results are presented in Table 1 in which T_ω denotes the execution time of our algorithm, T_M denotes the execution time of the best baseline algorithm, and #proc is the number of procedures in each benchmark. The *speedup* column indicates the speedup of our method which is computed as T_M/T_ω. In the final row, all numbers in each column are the sum of the numbers in 7 benchmarks except the final number in the *speedup* column. The speedup in the final row is obtained from the values of T_M and T_ω in the final row which gives us the average speedup for the entire experiment.

Table 1 illustrates that we improved the performance in the incremental computation of WCC significantly. The performance improvements are between 2 to 4 times for smaller benchmarks compared to the best baseline approach. However, for larger benchmarks like *Perlbench* or *GCC*, the performance improvements are significant. We obtained a speedup of 9.35 and 35.29 times for *Perlbench* and *GCC* respectively. On average, we obtained a speedup of 31.03 for the entire experiment. This proves that our approach can be the best alternative to compute WCC over the state-of-the-art algorithms.

As seen in Table 1, the performance improvement results are skewed towards larger benchmarks such as *GCC* and *Perlbench*. We bring our discussion of these

skewed results from our earlier results on non-incremental WCC computation methods [14]. The higher gain for GCC is due to the fact that GCC is the largest benchmark in the benchmark suite, the size of CFGs for the procedures in this benchmark is much bigger than the size of the CFGs in the benchmarks like *Mcf*, *Xz*, or *Nab*. The *Xz* benchmark (for example) 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. *GCC* has 38 times more procedures than Xz and the average size of a CFG per procedure is 20. Also, greater speedups are obtained in larger CFGs. There are 171 and 55 procedures in *GCC* with the size 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. The *Perlbench* is the second largest benchmark in our experiments in terms of the CFG size and the number of procedures. We obtained the second highest speedup (9.35) for this benchmark. Our results would have been less skewed if we would consider other benchmarks similar to *GCC* or *Perlbench* that have more procedures and the CFG sizes are larger, unlike *Xz* or *Mcf*.

5 Related Work

The concept of control dependence was first introduced by Denning and Denning [5] in analyzing the information-flow security of programs. He used the dominator-based (inverse of postdominator) approach to identify program instructions influenced by the conditional instructions in the program. Weiser [22] shown how to use this concept in program slicing. This concept was first formalized by Ferrante et al. [6] and used it to compute the program dependence graph (PDG). PDG is a program representation that can be used for program slicing and program optimization techniques. This formal definition of control dependence was based on computing postdominator relation which is still being used in modern compilers such as LLVM or GCC for program transformation and program optimization techniques.

Several alternatives to this standard definition are introduced in the literature. The earliest of these alternatives is the work of Podgurski and Clarke [19]. They provided the concept of weak and strong syntactic dependence where strong syntactic dependence is the standard control dependence relation of Ferrante et al. and the weak syntactic dependence is the nontermination sensitive control dependence relation. Bilardi and Pingali [2] provided a generalized framework of Podgurski and Clarke which is parameterized with respect to a set of CFG paths providing different control dependence relations. The above control dependences were extended by Ranganath et al. [21,20] to deal with programs containing exceptions or nonterminating programs. Modern software such as web services, distributed systems, or robot control software may be nonterminating and it may be desirable to compute control dependence from such systems. The authors in Reference [21,20] introduced control dependencies that are applicable to these modern programming language constructs. They defined the nonter-

mination insensitive and nontermination sensitive control dependencies in the opposite sense of Podgurski and Clarke, and provided algorithms to compute these control dependence relations.

Danicic et al. [4] provided the concept of weak and strong control closure that are nontermination insensitive and nontermination sensitive respectively. These definitions are the most general and unifying definitions capturing a wide variety of programming language constructs. They have shown that all previously defined control dependence relations are the special case of these two generalized concepts. They have provided algorithms to compute WCC and SCC and the worst-case time complexity of these algorithms are $O(|N|^3)$ and $O(|N|^4)$ where $|N|$ is the number of vertices of the CFG.

More recently, a number of works extending and improving various concepts of Danicic et al. have been introduced. Our earlier works in [14,12] provided algorithms to compute WCC and SCC that improved the theoretical worst-case time complexity by an order of the size of the CFG as well as the practical efficiency. We have extended the definitions of WCC and SCC for interprocedural programs in order to prove the semantic correctness of dependence-based program slicing in [17], and provided an algorithm to compute the WCC for interprocedural programs in [18]. However, none of these improvements considered the incremental computation of WCC.

Léchenet et al. [10] provided an improvement of Danicic et al. by applying various optimizations and demonstrated the efficiency improvements in practical evaluation. The theoretical complexity of their algorithm is not provided and the algorithm is not incremental in nature. Khanfar et al. [8] developed a demand-driven algorithm to compute direct control dependencies to a particular program statement which requires that the program must have a unique exit point. This algorithm is not incremental in nature and their algorithm does not compute WCC.

Recently, we have shown an interesting duality relationship between computing the SSA program and the WCC relation [13]. Our incremental algorithm may provide an improved algorithm to compute the SSA program without computing the standard dominance frontier-based SSA construction as done in [15,16].

6 Conclusion and Future works

Numerous definitions of control dependency relations are introduced in the literature to handle a wide spectrum of programming language constructs. The weak and strong control closures are the most generalized definitions capturing nontermination (in)sensitive control dependencies. Since the introduction of these concepts, a series of works have been published to provide an improved algorithm computing WCC and SCC and extend them to handle interprocedural programs. However, there exists no effort that provides an incremental computation of WCC. Incremental computation of WCC is especially important for the client application of WCC such as program slicing since it requires the repeated computation of WCC in a fixpoint iteration. A non-incremental algorithm loses

performance by repeatedly computing unnecessary information. In this paper, we have developed a novel algorithm to compute WCC incrementally which is also the fastest algorithm among all the existing approaches to computing WCC. We have provided the proof of correctness of our method and analyzed its theoretical worst-case time complexity which is quadratic in terms of the size of the CFG. We have implemented our algorithm in the Clang/LLVM compiler framework and compared it with the best baseline approach by running experiments on well-known benchmarks. We have obtained an average speedup of 31.03 in all benchmarks and a maximum speedup of 35.29 in the largest benchmark. This gives us an indication that the amortized complexity of our algorithm is much better than the theoretical worst-case complexity.

The future direction of this work includes developing a method that also computes SCC and is applicable to interprocedural programs. The further extension will be to develop definitions and algorithms to handle time-sensitive weak and strong control closure that will be beneficial to detect timing leaks in security-critical software.

Acknowledgment This research is supported by the Swedish Knowledge Foundation via the HERO project

References

1. Amtoft, T.: Correctness of practical slicing for modern program structures. Tech. rep., Department of Computing and Information Sciences, Kansas State University (2007)
2. Bilardi, G., Pingali, K.: A framework for generalized control dependence. SIGPLAN Not. **31**(5), 291–300 (May 1996). https://doi.org/10.1145/249069.231435
3. Bucek, J., Lange, K.D., v. Kistowski, J.: Spec cpu2017: Next-generation compute benchmark. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. pp. 41–42. ICPE '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3185768.3185771
4. Danicic, S., Barraclough, R., Harman, M., Howroyd, J.D., Kiss, Á., Laurence, M.: A unifying theory of control dependence and its application to arbitrary program structures. Theoretical Computer Science **412**(49), 6809–6842 (2011). https://doi.org/10.1016/j.tcs.2011.08.033
5. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (Jul 1977). https://doi.org/10.1145/359636.359712
6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: Dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (Jul 1987). https://doi.org/10.1145/24039.24041
7. Khanfar, H., Lisper, B., Masud, A.N.: Static backward program slicing for safety-critical systems. In: de la Puente, J.A., Vardanega, T. (eds.) Reliable Software Technologies – Ada-Europe 2015. Lecture Notes in Computer Science, vol. 9111, pp. 50–65. Springer (2015). https://doi.org/10.1007/978-3-319-19584-1_4
8. Khanfar, H., Lisper, B., Mubeen, S.: Demand-driven static backward slicing for unstructured programs. Tech. rep. (May 2019), `http://www.es.mdh.se/publications/5511-`

9. Lattner, C., Adve, V.: The LLVM Compiler Framework and Infrastructure Tutorial. In: LCPC'04 Mini Workshop on Compiler Research Infrastructures. West Lafayette, Indiana (Sep 2004). https://doi.org/10.1007/11532378_2
10. Léchenet, J.C., Kosmatov, N., Le Gall, P.: Fast computation of arbitrary control dependencies. In: Russo, A., Schürr, A. (eds.) Fundamental Approaches to Software Engineering. pp. 207–224. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_12
11. Lisper, B., Masud, A.N., Khanfar, H.: Static backward demand-driven slicing. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 115–126. PEPM '15, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2678015.2682538
12. Masud, A.N.: Simple and efficient computation of minimal weak control closure. In: Pichardie, D., Sighireanu, M. (eds.) Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12389, pp. 200–222. Springer (2020). https://doi.org/10.1007/978-3-030-65474-0_10
13. Masud, A.N.: The duality in computing ssa programs and control dependency. IEEE Transactions on Software Engineering pp. 1–16 (2022). https://doi.org/10.1109/TSE.2022.3192249
14. Masud, A.N.: Efficient computation of minimal weak and strong control closure. J. Syst. Softw. **184**, 111140 (2022). https://doi.org/10.1016/j.jss.2021.111140
15. Masud, A.N., Ciccozzi, F.: 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. pp. 23–33. IEEE (2019). https://doi.org/10.1109/SCAM.2019.00012
16. Masud, A.N., Ciccozzi, F.: More precise construction of static single assignment programs using reaching definitions. Journal of Systems and Software **166**, 110590 (2020). https://doi.org/https://doi.org/10.1016/j.jss.2020.110590
17. Masud, A.N., Lisper, B.: Semantic correctness of dependence-based slicing for interprocedural, possibly nonterminating programs. ACM Trans. Program. Lang. Syst. **42**(4) (Jan 2021). https://doi.org/10.1145/3434489
18. Masud, A.N., Lisper, B.: On the computation of interprocedural weak control closure. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. p. 65–76. CC 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3497776.3517782
19. Podgurski, A., Clarke, L.A.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng. **16**(9), 965–979 (Sep 1990). https://doi.org/10.1109/32.58784
20. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. In: European Symposium on Programming. LNCS, vol. 3444, pp. 77–93. Springer-Verlag (2005). https://doi.org/10.1007/978-3-540-31987-0_7
21. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. **29**(5) (Aug 2007). https://doi.org/10.1145/1275497.1275502
22. Weiser, M.: Program slicing. In: Proc. 5th International Conference on Software Engineering. pp. 439–449. ICSE '81, IEEE Press, Piscataway, NJ, USA (1981), http://dl.acm.org/citation.cfm?id=800078.802557