A Framework for Path Sensitive Program Analysis

Vijayaraghavan Murali
National University of Singapore
m.vijay@nus.edu.sg

Joxan Jaffar   Jorge A. Navas
National University of Singapore
{joxan,navas}@comp.nus.edu.sg

Andrew E. Santosa
University of Sydney, Australia
santosa@it.usyd.edu.au

Abstract

We present a framework that produces path-sensitive analyses with different tradeoffs of accuracy and efficiency. The first component is a program transformation that restructures a CFG in order to encode path-sensitivity into it. The method consists of deleting infeasible paths from the CFG while performing selective *node splitting* based on information captured from infeasible paths. This transformation is fully independent from the analysis and can be built *offline*. Our initial experiments demonstrate that the size of the resulting CFG increases by a reasonable factor and its use can produce significant accuracy gains for several analyses.

The second component is a generic backward algorithm that interleaves the above process with the computation of the analysis. This synergy allows using analysis information in order to decide whether a node should be *joined* or not. We use the concept of *witness* that establishes the conditions, using some knowledge from the analysis, to ensure that a node can be joined without incurring in any loss of accuracy. We demonstrate that although more expensive this concept can be implemented producing more precise results.

1. Introduction

Program analyses that directly use a Control Flow Graph (CFG) or similar program representation often incur in two kind of loss of accuracy: inclusion of *infeasible paths*, and merging of different abstract states (via join operator) along incoming edges of a *control flow merge*. Although these over-approximations are often precise enough to reason about the property of interest, they may make compiler optimizations unable to be applied or raise false alarms in program verification and testing. In that case, the analysis designer might need to modify the analysis which is often far from being trivial.

We present a framework for constructing path-sensitive program analyses. The first component of our framework consists of a program transformation that given a CFG produces another CFG with path-sensitiveness encoded into it, the *Path Sensitive Control Flow Graph (PSCFG)*. A PSCFG poses the following interesting properties:

(1) it excludes infeasible paths, and

(2) merging points where the join operator may incur a loss of precision are split and their successors are duplicated.

We will show that (1) and (2) can be achieved at the expense of a reasonable increase in the size of the CFG.

There are two major features that make the PSCFG an object of general interest for any analysis designer:

- it is independent of the analysis, and thus, it can be built offline
- it can be used to enhance an arbitrary program analysis, and hence, does not impose any restriction on the analysis.

The second component of our framework is a generic algorithm which may further improve the precision provided by the PSCFG. The node splitting technique used during the construction of the PSCFG may be refined if some information from the analysis is known. Although this makes the algorithm dependent on the analysis, all steps are still generic (i.e., parameterized by the analysis) and hence, it significantly eases the burden of writing a path-sensitive analysis from scratch.

The main ingredients of our approach are a mix of symbolic execution, automatic loop invariants, and interpolants.

Symbolic execution [17] uses *symbolic values* as inputs instead of actual data and represents the values of program variables as symbolic expressions and functions of the input symbolic values. A *path condition* is maintained for each path and it is a formula over the symbolic inputs formed by accumulating constraints which those inputs must satisfy in order for execution to follow that path. A path is *infeasible* if its path condition is unsatisfiable. Otherwise, the path is *feasible*. A *symbolic execution tree* depicts all executed (feasible) paths during the symbolic execution.

The central idea is to run symbolic execution on the CFG while building a symbolic execution tree which resembles the final PSCFG. There are two main challenges: infinite length of symbolic paths and exponential number of symbolic paths. We follow [24] to automatically compute *loop invariants*. Because invariants are, in general, approximate our tree (which in fact is a graph due to loop abstractions) cannot be exact. Nevertheless our results in Sec. 5 demonstrate that our approach can still produce significant accuracy gains.

Then, we mitigate the path explosion problem as follows. Whenever an infeasible path is encountered we extract a *Craig interpolant* [8] which is a formula that preserves the infeasibility of the path. The purpose of an interpolant is to avoid the exploration of any path whose formula associated with its symbolic state entails an interpolant previously computed. We call this step the *subsumption test*.

Whenever a subsumption test does not hold, symbolic execution will naturally perform a node splitting and duplicate all its successors until the next merging point. Therefore, the key insight is that we can rely on the outcome of the *subsumption* tests in order to decide whether merging nodes should be split or not. Informally, if a node is subsumed by another node then the set of feasible paths reachable from the subsumed node is a *subset* of the set of feasible paths reachable from the subsumer. Thus, for any analysis executed on the PSCFG it is always safe for the subsumed node to *reuse* the analysis answers from the subsumer, thereby avoiding node splitting. This method prunes the search space keeping the size of the tree manageable.

Nevertheless, this over-approximation can still lead to sources of imprecision. Therefore, we introduce the second component of our framework: a generic algorithm that interleaves the above symbolic execution process with the execution of the analysis. The main objective is to refine the subsumption test with information from the analysis in order to avoid merging nodes when the analysis could lose precision, something that our analysis-independent program transformation could not do before.

We then present the concept of a *witness* that represents the conditions necessary to reuse analysis answers without any loss of precision. We claim that often the designer needs only to provide the analysis-specific operations (e.g., join operator and transfer function). Therefore, it saves designers the laborious task of implementing their path-sensitive analyzers from scratch. Although more expensive, this second component may pay off by producing more precise results for analysis with special accuracy requirements.

An important property of this generic algorithm is that it produces *exact* results for loop-free programs. By "exact" we mean the analysis cannot produce solutions from spurious (i.e., non-executable) paths[1]. For programs with unbounded loops, symbolic execution cannot be exact due to the use of loop invariants (and hence, consideration of some infeasible paths). Note that the fact that the concrete semantics cannot be inferred without computing the best abstract transfer function is considered an orthogonal issue. That is, our concept of *exactness* is independent from the precision of the abstract transfer function.

Organization. The rest of this paper is organized as follows. Sec. 2 describes the related work. Sec. 3 describes the formalism and definitions used in this paper. Sec. 4 presents the first component of our framework: a transformation that converts CFG into PSCFG. Sec. 5 demonstrates that our transformation can enhance several program analyses building PSCFGs of manageable sizes. Sec. 6 presents the second component: a generic algorithm that uses the concept of witness to improve further the precision of PSCFGs, its implementation, and preliminary results. Finally, Sec. 7 concludes.

2. Related Work

We center our discussion here to the most relevant works that take into account path-sensitiveness to enhance program analysis. We also discuss uses of interpolation that might have influenced our work.

Similar to [13, 19] our algorithm discovers invariant interpolants that preserve the infeasibility of the paths. This is where the similarity ends. Those works focus on proving the unreachability of certain error nodes while ours focuses on enhancing program analyses in order to improve their accuracy.

[12] uses path-sensitiveness inherent in CEGAR to improve precision of certain kind of dataflow analyses. Other works with similar spirit are ESP [9] and [10]. The former keeps track of some branch correlations under the assumption that different branches that produce different results should be treated differently. The latter improves ESP by adjusting the criterion at merge points where dataflow analysis loses precision. We differ from this line of works in a very clear way. Our framework does not use counterexamples in order to achieve path-sensitiveness. However, these works must have a target property to generate those counterexamples. As a result, our PSCFG, for instance, can be used for enhancing a richer set of dataflow analyses that includes live variable analysis, alias analysis, slicing, reaching definitions, constant propagation, etc.

[1] Of course, limited by theorem prover technology which decides whether a path condition is unsatisfiable or not.

Profiling techniques [2] identify those paths more frequently visited during the execution of multiple tests ("hot paths"). Join points in the CFG that belong to hot paths are split if the dataflow analysis may incur in a loss of precision. This technique is dynamic while ours is static. Hence, we could complement each other. Similar to us [25] propose a method to restructure a given CFG into one that is path sensitive, but there are two key differences with our PSCFG: they do not deal with infeasible paths and their node splitting is based on whether the merge is destructive or not (i.e., if the analysis may lose information during the join operator), which makes their approach dependent on the analysis.

The design of path-sensitive analyses has been and remains a very active area of research. Bodik et al. [3–5] describe several dataflow analyses improved by detecting infeasible paths through branch correlation. [23] present an improved WCET analysis by eliminating infeasible paths detected using conflict sets. [22] describe a backward slicer that refines a sliced PDG (Program Dependency Graph) [14] by eliminating dependencies between nodes along infeasible paths. They use a different concept of path condition. They keep track of necessary conditions for information flow between two points in the PDG, that is, conditions that must hold for information flow to occur between those two points. Thus, their notion of path condition is already abstracted. Moreover, they use Binary Decision Diagrams (BDDs) to overcome the potential combinatorial explosion. BDDs and interpolation are orthogonal concepts. BDDs may provide a more compact representation of the symbolic execution tree. However, interpolation allows pruning the search space which may provide more significant savings. *Trace Partioning* [18] is an abstract domain to decide whether or not merge the abstract states at the join points in the CFG. However, [18] is a theoretical description of the domain and hence, it does not address practical issues.

Finally, our closest related work is [15]. They present an interpolation-based dynamic programming algorithm to solve a combinatorial optimization problem: the Resource-Constrained Shortest Path (RCSP) problem. This work can be seen as an instance of the second component of our framework (Sec. 6). Moreover, this problem is simpler in the sense it is defined for a finite setting while our framework handles loops.

3. Background

Syntax. We restrict our presentation to a simple imperative programming language [2], where all basic operations are either assignments or assume operations, and the domain of all variables are integers. The set of all program variables is denoted by *Vars*. An *assignment* x := e corresponds to assign the evaluation of the expression e to the variable x. In the *assume* operator, assume(c), if the boolean expression c evaluates to *true*, then the program continues, otherwise it halts. The set of operations is denoted by *Ops*.

We model a program by a *transition system*. A transition system is a quadruple $\langle \Sigma, I, \longrightarrow, O \rangle$ where Σ is the set of states and $I \subseteq \Sigma$ is the set of initial states. $\longrightarrow \subseteq \Sigma \times \Sigma \times Ops$ is the transition relation that relates a state to its (possible) successors executing operations. This transition relation models the operations that are executed when control flows from one program location to another. We shall use $\ell \xrightarrow{\mathsf{op}} \ell'$ to denote a transition relation from $\ell \in \Sigma$ to $\ell' \in \Sigma$ executing the operation $\mathsf{op} \in Ops$. Finally, $O \subseteq \Sigma$ is the set of final states.

Symbolic Execution. A *symbolic state* υ is a triple $\langle \ell, s, \Pi \rangle$. The symbol $\ell \in \Sigma$ corresponds to the current program counter (with special program counters for initial, ℓ_{start}, and final locations, ℓ_{end}). The symbolic store s is a function from program variables to terms

[2] Our implementation supports most features of sequential C including function calls and pointers.

over input symbolic variables. Each program variable is initialized to a fresh input symbolic variable. The *evaluation* $[\![e]\!]_s$ of an arithmetic expression e in a store s is defined as usual: $[\![v]\!]_s = s(v)$, $[\![n]\!]_s = n$, $[\![e+e']\!]_s = [\![e]\!]_s + [\![e']\!]_s$, $[\![e-e']\!]_s = [\![e]\!]_s - [\![e']\!]_s$, etc. The evaluation of Boolean expression $[\![b]\!]_s$ can be defined analogously. Finally, Π is called *path condition* and it is a first-order formula over the symbolic inputs and it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular corresponding path. The set of first-order formulas and symbolic states are denoted by *FO* and *SymStates*, respectively. Given a transition system $\langle \Sigma, I, \longrightarrow, O\rangle$ and a state $\upsilon \equiv \langle \ell, s, \Pi\rangle \in SymStates$, the symbolic execution of $\ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state υ' defined as:

$$\upsilon' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!]_s\rangle & \text{if } \mathsf{op} \equiv \mathsf{assume(c)} \text{ and } \Pi \wedge [\![c]\!]_s \\ & \text{is satisfiable} \\ \langle \ell', s[x \mapsto [\![e]\!]_s], \Pi\rangle & \text{if } \mathsf{op} \equiv \mathsf{x := e} \end{cases} \quad (1)$$

Note that Eq. (1) queries a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound but not complete. That is, the theorem prover must say a formula is unsatisfiable only if it is indeed so.

Abusing notation, given a symbolic state $\upsilon \equiv \langle \ell, s, \Pi\rangle$ we define $[\![\upsilon]\!] : SymStates \rightarrow FO$ as the projection of the formula $(\bigwedge_{v \in Vars} [\![v]\!]_s) \wedge [\![\Pi]\!]_s$ onto the set of program variables *Vars*. The projection is performed by the elimination of existentially quantified variables.

A *symbolic path* $\pi \equiv \upsilon_0 \cdot \upsilon_1 \cdot \ldots \cdot \upsilon_n$ is a sequence of symbolic states such that $\forall i \bullet 1 \leq i \leq n$ the state υ_i is a *successor* of υ_{i-1}. A symbolic state $\upsilon' \equiv \langle \ell', \cdot, \cdot\rangle$ is a successor of another $\upsilon \equiv \langle \ell, \cdot, \cdot\rangle$ if there exists a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$. A path $\pi \equiv \upsilon_0 \cdot \upsilon_1 \cdot \ldots \cdot \upsilon_n$ is *feasible* if $\upsilon_n \equiv \langle \ell, s, \Pi\rangle$ such that $[\![\Pi]\!]_s$ is satisfiable. If $\ell \in O$ and υ_n is feasible then υ_n is called *terminal* state. Otherwise, if $[\![\Pi]\!]_s$ is unsatisfiable the path is called *infeasible* and υ_n is called *infeasible* state. A state $\upsilon \equiv \langle \ell, \cdot, \cdot\rangle$ is called *subsumed* if there exists another state $\upsilon' \equiv \langle \ell, \cdot, \cdot\rangle$ such that $[\![\upsilon]\!] \models [\![\upsilon']\!]$. If there exists a feasible path $\pi \equiv \upsilon_0 \cdot \upsilon_1 \cdot \ldots \cdot \upsilon_n$ then we say υ_k $(0 \leq k \leq n)$ is *reachable* from υ_0 in *k steps*. We say υ'' is reachable from υ if it is reachable from υ in some number of steps.

A *symbolic execution tree* depicts the execution paths followed during the symbolic execution of a transition system by triggering Eq. (1). The nodes represent symbolic states and the arcs represent transitions between states. We say a symbolic execution tree is *complete* if it is finite and all its leaves are either terminal, infeasible or subsumed.

Abstract Interpretation. An abstract domain $\mathcal{A}$ is defined as a lattice structure $\langle \mathcal{L}, \sqsubseteq, \bot, \sqcup, \sqcap, \top\rangle$ partially ordered by $\langle \mathcal{L}, \sqsubseteq\rangle$ where $\sqcup$ is the *least upper bound* and $\sqcap$ is the greatest lower bound operators. The symbols $\bot$ and $\top$ are the least and greatest element, respectively. Additionally $\mathcal{A}$ can be equipped with widening ∇ and narrowing $\triangle$ operators. We assume $\mathcal{A}$ is Galois-connected [6] with the powerset lattice of state sets, and the use of α and γ for the abstraction and concretization maps for this Galois connection.

Program Analyses. We define a program analysis by using the classical *Monotone Framework* [1] which consists of:

- A lattice $\mathcal{L}$ partially ordered by $\sqsubseteq$ with $\bot$ of finite length and an initial value $\sigma_{\mathsf{init}} \in \mathcal{L}$.
- A transfer function f [3] from $\mathcal{L}$ to $\mathcal{L}$. We demand each transfer function is at least *monotone* (i.e., $f(l \sqcup l') \sqsubseteq f(l) \sqcup f(l')$). The definition of this function depends on the direction of propagation of information: forward ($\widehat{post} : \mathcal{L} \times Ops \rightarrow \mathcal{L}$) or backward ($\widehat{pre} : \mathcal{L} \times Ops \rightarrow \mathcal{L}$). Intuitively, $\widehat{pre}(\sigma_{post}, \mathsf{op})$ ($\widehat{post}(\sigma_{pre}, \mathsf{op})$) returns the *pre-state* (*post-state*) after executing backwards (forward) the operation op on the lattice value σ_{post} (σ_{pre}).
- A binary join [4] operation $\sqcup$, on $\mathcal{L}$, to represent the *confluence operator* (i.e., combine information from different paths).

[3] The monotone framework defines a set of transfer functions such that each different location or block may use a different transfer function. For simplicity, we assume a single transfer function f.

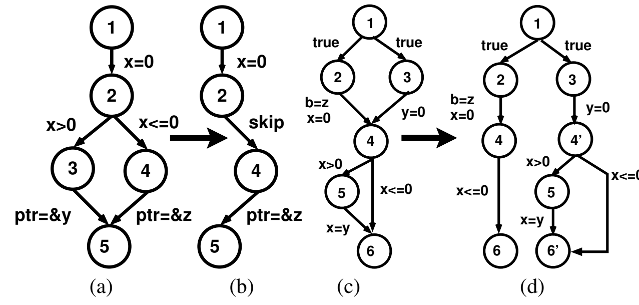

Figure 1. Deletion of Infeasible Paths and Node Splitting

4. Path-Sensitive Control Flow Graph (PSCFG)

Analyses that use directly a Control Flow Graph (CFG) often incur in two kind of loss of accuracy: consideration of *infeasible paths*, and merging of different abstract states via confluence operator. The first component of the framework is a program transformation that restructures the original CFG[5] to alleviate those sources of imprecision by:

(Rule 1) eliminating infeasible paths

(Rule 2) splitting explicitly merging points in case a loss of precision was possible during the confluence (join) operator of the analysis.

The result of this transformation is another CFG, the so called *PSCFG*, with the following features:

- it is encoded with *path-sensitivity* which might make an arbitrary *off-the-shelf* program analysis more precise. This is demonstrated experimentally in Sec. 5.
- it is *independent* to the analysis, and thus, it can be built offline
- at the expense of a *reasonable increase* in the size with respect to the original CFG (see also Sec. 5 for empirical demonstration).

The potential benefits of Rule 1 are quite obvious. The elimination of spurious paths may eliminate some imprecision during the execution of program analyses. Consider the CFG in Fig. 1(a). The nodes are labeled with program locations and edges between two locations labeled by the instruction that executes when control moves from the source to the destination. The path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$ is not executable due to the unsatisfiability of the constraints $x = 0 \wedge x > 0$. Therefore, we could transform the original CFG into the one shown in Fig. 1(b), by removing the edge where unsatisfiability was detected and all its successors up to next merging point (i.e., 5). Then, assume we would like to run an Andersen-like pointer analysis on the program in Fig 1(a). The analysis will report that ptr,&y, and &z may point to the same memory location. However, it is straightforward to see that the same analysis on the transformed CFG cannot infer that ptr and &y are aliased producing a more accurate result.

[4] Classical literature says often a "meet" operator. This is because classical literature focus on analyses where $\sqcup$ is $\bigcap$.

[5] Here, and in the rest of the paper, a CFG and transition system as defined in Sec. 3 are interchangeable terms since we can convert trivially one into another.

The logic behind Rule 2 is also related to the existence of infeasible paths but in a more elaborated way. Consider now the CFG in Fig. 1(c). Note that this CFG also poses an infeasible path $1 \to 2 \to 4 \to 5 \to 6$ due to the unsatisfiability of $x = 0 \wedge x > 0$. The difference here wrt to Fig. 1(a) is that we cannot eliminate the edges $4 \to 5$ and $5 \to 6$ since there is another path, $1 \to 3 \to 4 \to 5 \to 6$, exploring those edges. However, we can split node 4, and subsequently duplicate all its successors up to the next merging point while deleting infeasible paths whenever possible as illustrated in Fig. 1(d). Now, consider a live variable analysis (backward analysis that over-approximates the set of variables that can be used in the future before their next definition [1]). In Fig. 1(c), the set of live variables at node 4 is $\{x,y\}$. In the PSCFG in Fig. 1(d) the set of live variables is $\{x\}$ at node 4 and $\{x,y\}$ at new node $4'$. Note what happens now. After the execution of the transfer function $\widehat{pre}$[6] on $3 \to 4'$ the variable y is not alive anymore since it is defined. As a result, the live variables at 3 in the PSCFG is only $\{x\}$. On the other hand, the live variables at node 2 is simply $\{z\}$. Therefore, the final set of live variables at node 1 is the set union $\{x,z\}$, as opposed to the same analysis on the CFG in Fig. 1(c) where the final set is $\{x,y,z\}$.

At this point, one could be tempted to build a symbolic execution tree as defined in Sec. 3 from a given CFG, and then, to perform the program analysis of interest on the symbolic execution tree rather than the original CFG. The benefits of using the symbolic execution tree are easy to see: it does not pose infeasible paths, and all merging points are explicitly split.

Unfortunately, the symbolic execution tree described so far cannot be built, in general.

The first problem is the infinite length of the symbolic paths due to the existence of unbounded loops. We overcome this fundamental problem following the approach of [24] which uses abstract interpretation [6] to automatically compute *approximate loop invariants*. Because invariants are, in general, approximate our symbolic execution tree cannot be exact. Nevertheless, our results in Sec. 5 demonstrate that our approach can still produce significant improvement for real programs.

Second, even if the symbolic execution tree is finite the number of symbolic paths is exponential in the number of program branches. Therefore, a naive approach of splitting all merging points is not tractable. A key observation is that it is not necessary to split all the merging points since *many symbolic paths have the same impact on the property of interest*. Since both Rules 1 and 2 describe transformations using only infeasibility information (due to our desire of being independent from analysis) the knowledge that we can extract from infeasible paths plays a key role in the decision whether to split or not.

Whenever the symbolic execution encounters an infeasible path it extracts a *unsatisfiable core* formula called *interpolant* [8] that still explains the reason of infeasibility. The purpose of interpolants is twofold:

A it mitigates the path explosion problem by halting the symbolic execution of those paths that *entail* (are *subsumed* by) the interpolant, and

B Rule 2 (i.e., splitting nodes) is only applied at a given merging point if the current symbolic state does not entail any interpolant computed previously for that particular merging point.

In other words, interpolants control both the size of our PSCFG and its shape. The rationale behind (A) is that an interpolant is a formula that preserves the infeasibility of a set of paths. Then, if a symbolic state entails that formula this means that the set of paths

```
PSTransf($\upsilon_k, \mathcal{P}$)
INPUT:  $\upsilon_k$: symbolic state, $\mathcal{P}$: transition system   /* CFG */
OUTPUT: $\mathcal{P}_{out}$: transition system   /* PSCFG */

let rec IntpSymExec($\upsilon_k \equiv \langle \ell, s, \Pi \rangle, \mathcal{P}, \mathcal{M}, \mathcal{E}$)
1:  if $[\![\Pi]\!]_S$ is unsat then
2:    let $\mathcal{E}' = \mathcal{E} \setminus \{\_ \to \ell : k\}$   /* delete edge whose target is $\ell : k$ */
3:    return $\langle \mathcal{M} \cup \{\langle \ell, k \rangle : \textsc{Interp}([\![\upsilon_k]\!], \mathit{false})\}, \mathcal{E}' \rangle$
4:  else if $(\ell = \ell_{\mathsf{end}})$ then
5:    return $\langle \mathcal{M} \cup \{\langle \ell, k \rangle : \mathit{true}\}, \mathcal{E} \rangle$
6:  else if $\exists \langle \ell', k' \rangle : \overline{\Psi} \in \mathcal{M}$ s.t. $(\ell = \ell')$ and $([\![\upsilon_k]\!] \models \overline{\Psi})$ then
7:    return $\langle \mathcal{M} \cup \{\langle \ell, k \rangle : \overline{\Psi}\}, \mathcal{E} \cup \{\ell : k \longrightarrow \ell' : k'\} \rangle$
8:  else if $\exists\, \ell \to \ell'$ such that $\ell' : k'$ is an ancestor in dfs then
9:    $\overline{\upsilon}_k$ := INVARIANT($\upsilon_k, \ell' \to \ldots \to \ell$)
10:   return $\langle \mathcal{M} \cup \{\langle \ell, k \rangle : [\![\overline{\upsilon}_k]\!]\}, \mathcal{E} \cup \{\ell : k \longrightarrow \ell' : k'\} \rangle$
11: else if $\ell$ is a loop header then
12:   $\upsilon_k$ := INVARIANT($\upsilon_k, \ell \to \ldots \to \ell$)
13:   goto 15
14: else
15:   $\overline{\Psi}$ := true
16:   foreach transition relation $\ell \xrightarrow{\mathsf{op}} \ell' \in \mathcal{P}$ do
17:     $\upsilon'_{k'} \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!]_S \rangle & \text{if op} \equiv \text{assume(c) (fresh } k'\text{)} \\ \langle \ell', s[x \mapsto [\![e]\!]_S], \Pi \rangle & \text{if op} \equiv \text{x := e (fresh } k'\text{)} \end{cases}$
18:     $\langle \mathcal{M}, \mathcal{E} \rangle$ := IntpSymExec($\upsilon'_{k'}, \mathcal{P}, \mathcal{M}, \mathcal{E} \cup \{\ell : k \xrightarrow{\mathsf{op}} \ell' : k'\}$)
19:     $\overline{\Psi}$ := $\overline{\Psi} \wedge (\bigwedge_{\langle \cdot, k' \rangle : \overline{\Psi}' \in \mathcal{M}} \widehat{wp}(\mathsf{op}, \overline{\Psi}'))$
20:   endfor
21:   return $\langle \mathcal{M} \cup \{\langle \ell, k \rangle : \overline{\Psi}\}, \mathcal{E} \rangle$
22: end
in
  $\langle \cdot, \mathcal{E} \rangle$ := IntpSymExec($\upsilon_k, \mathcal{P}, \emptyset, \emptyset$)
  build a transition system $\mathcal{P}_{out}$ from $\mathcal{E}$   /* trivial step */
  return $\mathcal{P}_{out}$
end
```

Figure 2. Program Transformation that Produces a PSCFG

emanating from that (subsumed) symbolic state cannot have less infeasible paths (or equivalently, more executable paths). Therefore, it is safe from the point of view of the program analysis if we halt the symbolic execution at the subsumed symbolic state. (B) is a by-product of (A). If a node (i.e., symbolic state) is subsumed by the interpolant of another node in the tree we merge the two nodes. Otherwise, we keep them separate forcing the node splitting.

We next formally introduce the two key concepts which will decide when the symbolic execution can be halted and whether it can split nodes or not.

DEFINITION 1 (Interpolant). *Given two first-order logic formulas Π_1 and Π_2 such that $\Pi_1 \wedge \Pi_2$ is unsatisfiable a Craig* interpolant *[8] is another first-order logic formula $\overline{\Psi}$ such that (a) $\Pi_1 \models \overline{\Psi}$, (b) $\overline{\Psi} \wedge \Pi_2$ is unsatisfiable, and (c) all variables in $\overline{\Psi}$ are common variables in Π_1 and Π_2.*

We then augment symbolic execution of a program by annotating each symbolic state with its corresponding interpolant such that the interpolant represents the sufficient conditions to preserve the infeasibility of the paths. Then, the notion of subsumption can be redefined as follows.

DEFINITION 2 (Subsumption with Interpolants). *Given two symbolic states υ and υ' such that υ is annotated with the interpolant $\overline{\Psi}$, we say that υ' is* subsumed *by υ if $[\![\upsilon']\!]$ implies $\overline{\Psi}$ (i.e., s.t. $[\![\upsilon']\!] \models \overline{\Psi}$).*

[6] $\widehat{pre}(\sigma, \mathsf{op}) = (\sigma \setminus def(\mathsf{op})) \cup use(\mathsf{op})$, where def and use are the defined and used variables in op.

A full description of the algorithm that takes a CFG (i.e., a transition system) and produces another CFG encoded with path-sensitivity (PSCFG) is given in Fig. 2. The input of the algorithm is an initial symbolic state $\upsilon_k \in \mathit{SymStates}$ and the transition system $\mathcal{P}$. We use the key k to refer unambiguously to the symbolic state υ in the symbolic execution tree. The edges of this symbolic execution tree are recorded in $\mathcal{E}$. In order to perform subsumption tests our algorithm maintains the table $\mathcal{M}$ that stores entries of the form $\langle \ell,k \rangle : \overline{\Psi}$, where $\overline{\Psi}$ is the interpolant at program location ℓ associated to a symbolic state k in the symbolic execution tree. The interpolants are generated by a procedure $\mathsf{Interp} : FO \times FO \to FO$ that takes two formulas and computes a Craig interpolant following Def. 1.

Essentially, the algorithm builds a symbolic execution tree executing $\mathcal{P}$ given the initial symbolic state while computing interpolants to prune redundant symbolic paths and performing selective abstractions via loop invariants in order to make finite the tree. During the symbolic execution of $\mathcal{P}$, Rule 1 and 2 are applied whenever possible as follows. If an infeasible path is detected then the symbolic execution is halted and hence, the rest of path is discarded from consideration. Moreover, an interpolant to preserve the infeasibility of the path is computed. Then, whenever a merging point is visited again by the symbolic execution we test whether the symbolic state is subsumed by the interpolant of any previous state that reached the merging point before. If yes, the symbolic execution halts the exploration of the path and joins it with the merging point. Otherwise, symbolic execution continues the execution of the path which implicitly forces the splitting of the node and the duplication of all its successors up to the next merging point. After termination, the set $\mathcal{E}$ contains the edges of the symbolic execution tree. This set of edges induces, in fact, a graph (possibly with cycles due to loop abstractions) which after some trivial post-processing can be used to build the output transition system $\mathcal{P}_{out}$ (i.e., PSCFG).

The algorithm PSTransf (Fig. 2) is defined in terms of the recursive function IntpSymExec. The base cases for each kind of leaf node in the symbolic tree are: infeasible, terminal, and subsumed. The recursive case unwinds the tree one level by executing one symbolic step. For clarity of presentation, let us omit the lines 8-13 for now, and assume programs do not have loops. Later, we shall describe how to handle loops.

The algorithm starts by testing if the path is infeasible at line 1. If yes, an interpolant is generated to avoid exploring again paths which have the same infeasibility reason (line 3). Note that the last (infeasible) edge is removed from $\mathcal{E}$ (line 2) and since symbolic execution stops the rest of the path will not be included in $\mathcal{E}$ (Rule 1). Next, the execution reaches the end of a path, line 4. The algorithm simply adds an entry in the *subsumption table* whose interpolant is *true* (line 5) since the symbolic path is feasible (i.e., there are no false paths to preserve) and returns the same $\mathcal{E}$. The main objective of lines 6-7 is to fully avoid path enumeration by searching for another state whose interpolant $\overline{\Psi}$ is entailed by the formula associated with current symbolic state υ_k. If the test holds, it returns the interpolant associated with the subsuming node (i.e., $\overline{\Psi}$), and since the entailment test was successful there is no reason to split the current (subsumed) node. Therefore, a new *unlabelled* edge, whose meaning is a no-op transition, from the subsumed state to the subsuming is added into $\mathcal{E}$. This is how we join the two states at the merging point. Otherwise, the symbolic execution must continue the exploration of the path which implicitly produces the split of the merging point (Rule 2), and the duplication of all its successors up to at least the next merging point.

In the remaining case, the symbolic execution continues forward one level in the symbolic execution tree. The foreach loop (lines 16-20) executes one symbolic step for each successor node [7] and it calls recursively again to IntpSymExec with each successor state (line 18). Once the recursive call returns the key remaining step is to compute an interpolant that generalizes the symbolic execution tree at the current node while preserving the infeasibility conditions of its successor nodes. The procedure $\widehat{wp} : Ops \times FO \to FO$ computes ideally the *weakest precondition (wp)* [11] which is the weakest formula on the initial state ensuring the execution of an operation in a final state, assuming it terminates. In practice, we approximate *wp* by making a linear number of calls to a theorem prover following the techniques described in [16]. The final interpolant $\overline{\Psi}$ added in the *subsumption table* is a first-order logic formula consisting of the *conjunction* of the result of $\widehat{wp}$ on each child's interpolant (line 19).

Loops. We continue describing our algorithm by discussing how it handles loops. The main challenge is to produce a finite symbolic execution tree. We use abstract interpretation to estimate loop invariants with the purpose of achieving finiteness. Here, we follow [24] in order to compute the least fixed point of the program using a numerical abstract domain and then, produce the desired loop invariants. Based on our initial experiments and our reduced set of benchmarks, a simple domain such as interval analysis has been enough.[8] Thus, we focus here on the interval analysis but same ideas can be easily applied to other abstract domains. We describe interval domain as usual: $I \triangleq \{\bot\} \cup \{[a,b] \mid (a,b \in \mathit{Int} \cup \{-\infty,\infty\}) \wedge a \leq b\}$ and computing the least fixed point employing wFix, on the lambda function:

$$F \equiv \lambda\, I \bullet \{\sigma' \mid \exists \sigma \in I \bullet \ell \xrightarrow{\mathsf{op}} \ell' \; \mathit{and} \; \sigma' = \widehat{post}_I(\sigma,\mathsf{op})\}$$

$$\mathsf{wFix}\; F \triangleq \left\{ \begin{array}{l} \textbf{let } a_0 = \bot \text{ and } a_{n+1} = a_n \nabla (F\ a_n) \\ \quad\; b_0 = \sqcup_n a_n \text{ and } b_{n+1} = b_n \triangle (F\ b_n) \\ \textbf{in } \sqcap_n b_n \end{array} \right.$$

to estimate the range by an integer interval[9]. As also described in [24], those ranges must be translated into first-order logic formulas. The function trst translates an abstract state into a formula:

$$\mathsf{trst}(\sigma_I) \triangleq \bigwedge_{x \in Vars} \mathsf{tr}(\sigma_I(x), x)$$

$$\mathsf{tr}([a,b],x) \triangleq (a \leq x) \wedge (x \leq b) \quad \mathsf{tr}(\bot,x) \triangleq \mathit{false}$$

Here, and in the rest of this paper, we will assume that when PSTransf is called, the input transition system has been previously preprocessed. This preprocessing consists of annotating the original CFG in such a way that program locations are labelled with the invariants inferred automatically by the abstract interpreter. We assume the abstract interpreter provides a function getAssrt which given a program counter ℓ, returns an assertion, in form of a first-order logic formula via trst, and renamed accordingly. Note that getAssrt already returns a loop invariant. However, we would like to strengthen it using the constraints propagated from the symbolic execution. The function INVARIANT performs this task as follows:

```
INVARIANT($\langle \ell, s, \Pi \rangle, \ell_1 \to \ldots \to \ell_n$) $\triangleq$
    let $s'$ := HAVOC($s$, MODIFIES($\ell_1 \to \ldots \to \ell_n$))
        $\overline{\Pi}$ := getAssrt($\ell$) $\wedge\, \Pi$
    in $\langle \ell, s', \overline{\Pi} \rangle$
```

[7] Note that the rule described in line 17 is slightly different from the one described in Sec. 3 because no satisfiability check is performed. Instead, this check is postponed and done by line 1.

[8] Of course, reasoning about other programs may need more refined analysis such as *octagon* for relations of the form $\overset{+}{-} v_i \overset{+}{-} v_j \leq c_{ij}$, *non-relational bitfield domain* to infer for each bit of an integer variable if its value is 0, 1 or may be either, and so on.

[9] We refer to [24] for definition of the narrowing and widening operators required for wFix and the rest of the details.

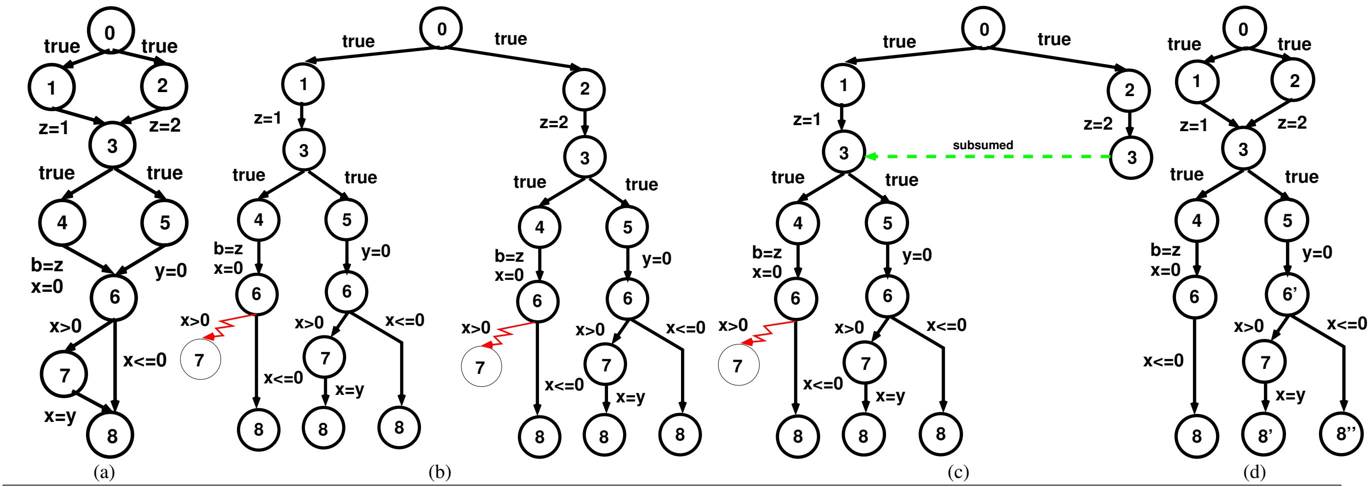

Figure 3. A CFG, Its Naïve Symbolic Tree, Its Interpolation-Based Symbolic Tree, and Its PSCFG

where $\textsc{Havoc}(s, Vars) \triangleq \forall v \in Vars \bullet s[v \mapsto S_z]$ where S_z is a fresh symbolic variable, and $\textsc{Modifies}(\ell_1 \to \ldots \to \ell_n)$ takes a sequence of transitions and it returns the set of variables that may be modified during its symbolic execution.

Let us now come back to the description of the function IntpSymExec. We first identify all the entry points (i.e., *loop header* [1]) of the loops. If we reach a loop header (line 11) then we abstract the current symbolic state by calling the procedure $\textsc{Invariant}$ (line 12) and continue with the exploration of the paths (line 13). The key difference is that the remaining symbolic execution will be performed with the (possible weaker) loop invariant computed by $\textsc{Invariant}$ rather than with the original (possibly stronger) symbolic state. The other important case is when symbolic execution encounters a *loop-back* edge at line 8. Here, we also need to produce an interpolant. For simplicity, we use the loop invariant as an interpolant (line 9). By doing this, it is straightforward to see that the symbolic state at the loop-back edge entails the state of the ancestor achieving *child-parent* subsumption in a straightforward manner. In practice, we attempt to first produce the interpolant from the exit paths of the loop and use them as the interpolants at the loop-back edges. This solution often produces weaker interpolants. Note that we also add a cycle in $\mathcal{E}$ by adding an unlabelled edge from the child to its parent.

To conclude with the explanation of our algorithm PSTransf we show how it executes symbolically the CFG in Fig 3(a) while producing the PSCFG of Fig 3(d).

Example 1. *For comparison, we first show the naive symbolic execution tree in Fig. 3(b). This tree can be obtained simply by executing exhaustively Eq.1 in Sec. 3. Since there is no loops, this process will terminate. Feasible transitions are denoted by (black) solid edges, and (red) infeasible transitions by zigzag edges. Now, consider the symbolic execution tree in Fig. 3(c) constructed by PSTransf.*

Say we execute first the path $0 \to 1 \to 3 \to 4 \to 6 \to 7$. The symbolic execution detects that the path condition at 7 is inconsistent since the symbolic state is of the form $\langle \ell_7, [z \mapsto 1, x \mapsto 0, b \mapsto 1, \ldots], false \equiv 0 > 0 \rangle$. At this point, the set $\mathcal{E}$ is $\{0{\to}1, 1{\to}3, 3{\to}4, 4{\to}6\}$[10] after removal of the infeasible edge $6 \to 7$. In addition, we generate the interpolant $x \leq 0$ at location 6 by weakest precondition. Then, we continue exploring two other paths through location 5 (i.e., $5 \to 6 \to 7 \to 8$ and $5 \to 6 \to 8$) which both are feasible. Note that we attempted unsuccessfully to halt the exploration of these paths at location 6, since the symbolic state at 6 did not entail the formula $x \leq 0$. As a consequence, the symbolic execution splits the node 6 and duplicates all its successors.

The algorithm then backtracks while generating interpolants and updating the set $\mathcal{E}$ until it explores the path $0 \to 2 \to 3$. Here again, we try to halt the symbolic execution. The interpolant for location 3 is true since $\widehat{wp}(b{=}z, x{=}0, x <= 0) = true$. Therefore, we can stop the symbolic execution since any formula entails true. As a result, symbolic execution joins the subsumed node 3 from $0 \to 2$ with the subsuming one from $0 \to 1$ avoiding splitting. We denote subsumed nodes in Fig. 3(c) by (green) dotted edges and the label "subsumed". The resulting path-sensitive CFG is shown in Fig. 3(d) after some trivial postprocessing. $\square$

We now state the correctness of the program transformation performed by PSTransf based on the *trace semantics* of our labelled transition system. Given a transition system $\langle \Sigma, I, \longrightarrow, O \rangle$, we use $\mathcal{F}(\Sigma)$ for the set of all finite non-empty sequences of states. A *trace* of a transition system $\mathcal{P}$ is an element of the set $\mathcal{P}\uparrow \triangleq \{s_0 \ldots s_n \in \mathcal{F}(\Sigma) \mid s_0 \in I, \forall\, i \bullet s_i \longrightarrow s_{i+1}\}$. Note that $\mathcal{P}\uparrow$ is the *prefix-closed* set of all finite traces of the program $\mathcal{P}$. Note that a particular execution of a program is a possible infinite path starting from s_0 such that there is no possible transition from the final state. We represent executions by the set of finite prefixes to avoid dealing with infinite paths.

Proposition 1. *Let $\mathcal{P}$ be a transition system and υ an initial state. Let Exact be the set of all finite non-empty, executable prefixes starting from υ in P. Then, if $\mathcal{P}^+ = PSTransf(\upsilon, \mathcal{P})$ then the following holds:* $$Exact \subseteq \mathcal{P}^+\uparrow \subseteq \mathcal{P}\uparrow$$

Note that Prop. 1 also states that the transition system P^+ returned by PSTransf cannot be less precise than the original transition system P since it will generally expose fewer sequences to the analysis.

Remark. Although our transformation is sound in approximating the concrete trace semantics of the original program, it does not ensure the preservation of certain properties related to the block structure of the original CFG. For instance, consider again the CFG in Fig. 3(a). Assume we would like to compute the postdominators[11] of the node 3. It is easy to see that the node 6 is a postdominator of 3 since all paths from 3 must pass through 6. Note that neither 4 and 5 postdominate 3. Now, let us look at the PSCFG in Fig. 3(d).

[10] For simplicity, we omit the labels.

[11] We say a node u in a CFG postdominates another node v if all paths from v to the exit node pass through u.

Benchmark	LOC	CFG Nodes	PSCFG Nodes	Ratio	Time
cdaudio	10245	2556	3041	1.19	3.9s
diskperf	5792	756	966	1.28	0.9s
fcron	6585	4473	8811	1.97	18.8s
floppy	7758	2803	3721	1.33	4.7s
kbfiltr	5685	533	711	1.33	0.6s
mpeg	2330	1447	2306	1.59	4.1s
pfinger	3595	1514	1785	1.18	2.1s
serial	9538	2650	3339	1.26	4.5s
statemate	1441	856	16720	19.53	144.0s
stunnel	7549	4698	7914	1.69	16.7s
tlan	3331	1672	2299	1.38	2.4s
Geom. Mean				**1.78**	
Average					**18.4s**

Figure 4. Size Growth and Construction Time of PSCFG

The original node 6 does not postdominate 3 anymore due to the existence of a duplicated node $6'$. Nevertheless, it is worth mentioning that this kind of anomalous behavior also arises in other node-splitting based program transformations such as in [25].

Fortunately, this does not preclude our transformation to be used by analyses in which dominance matters assuming some special considerations are taken into account. As an example, we focus on *program slicing* to illustrate how we can overcome this limitation. Weiser [27] defined a *backward slice* of a program wrt to a program point ℓ and a set of variables as all statements of the program that might affect the value of those variables at ℓ. To compute the slice, we need to keep track of two kind of dependencies: *data* and *control*. Data dependencies can be defined for a statement s by the dataflow equations $IN(s) = (OUT(s) \setminus \mathsf{def}(s)) \cup \mathsf{use}(s)$ if $\mathsf{def}(s) \cap OUT \neq \emptyset$. Otherwise, $IN(s) = OUT(s)$. The functions def and use refer to the set of defined and used variables in s. A statement s is in the slice if $IN(s) \cap \mathsf{def}(s) \neq \emptyset$. In addition, we need to propagate control dependencies. A guard g is included in the slice if any statement defined in any path from g up to its *nearest postdominator* is already in the slice.

Backward slicing can be done using PSCFG, the transition system returned by PSTransf, assuming we have also access to CFG, the original transition system. Data dependencies are computed on PSCFG. Since PSCFG may have multiple instances of the same statement s, we say that s is in the slice if at least one of the its instances is included. Although tedious, it is straightforward to keep relationships between any statement in CFG and any of its duplicates in PSCFG. Then, the remaining step is to propagate control dependencies by performing postdominance queries on CFG and then mapping back the results to PSCFG.

As a proof of concept, we have implemented a backward slicer 'a la' Weiser to demonstrate that our program transformation can also provide path-sensitiveness to static slicers in spite of the requirement of computing postdominance relationships. Moreover, our experimental evaluation in Sec. 5 shows that the use of PSCFG can significantly reduce the size of slices for real programs.

5. Experimental Evaluation of PSCFG

The main objectives of our experimental evaluation is to demonstrate, for a set of medium-size real programs, that:

(1) the size of the path-sensitive CFG (PSCFG) is often *manageable*. Our results show around a reasonable increasing of 78% wrt to the size of the original CFG.

(2) the use of PSCFG can improve significantly the results of several *off-the-shelf* program analyses. The accuracy gains vary from 10% to around 16%.

For our experiments, we used several device drivers which have previously been used in the verification and testing community: cdaudio, diskperf, floppy, kbfiltr, serial, and tlan. In addition we also used fcron, a cron daemon, statemate, a program generated by the STAtechart Real-time-Code generator STARC, mpeg, the mpeg-1 algorithm for compressing video, stunnel, a multiplatform SSL tunneling proxy and pfinger, a daemon for the standard finger protocol. All our experiments were carried out on an Intel 3.2GHz with 2Gb of memory.

We implemented a prototype of the program transformation described in Sec. 4 that produces PSCFGs. We model the heap as an array. A flow-insensitive pointer analysis is used to partition updates and reads into alias classes where each class is modeled by a different array. A theorem prover is used to decide linear arithmetic formulas over integer variables and array elements in order to check the satisfiability and entailment of formulas, and computing interpolants. The program is first annotated with approximate loop invariants produced by an abstract interpreter running the *interval analysis* [7]. Functions are analysed intraprocedurally, and hence, we do not perform function inlining. External functions are modeled as having no side effects and returning an unknown value.

Third party analyses generally take as inputs C programs rather than taking the CFG directly. Hence, in order for us to compare CFG and PSCFG, we decided to produce an equivalent C program from PSCFG, the so called *decompiled* program, and then, run the third party analyses on both the decompiled and original C programs for a fair comparison.

The decompilation process is quite straightfoward. Assignments and guards can be directly translated to C. Loops are decompiled using if-then-else and gotos. Functions are also straightforward since they are analyzed only once in an intraprocedural manner. Finally, joined nodes (subsumed nodes) are decompiled via gotos.

5.1 Size Growth and Construction Time of PSCFG

The main objective here is to evaluate the growth factor and total time in order to build a PSCFG. Our results are shown in Fig 4. The second column (LOC) represents the lines of code without comments in the benchmark. The third and fourth columns represent the number of nodes in the CFG and PSCFG, respectively. Note that these may not exactly correspond to the number of lines in the program because our prototype encodes blocks of consecutive assignments into a single node in the CFG.

The fifth column (Ratio) shows the ratio of PSCFG Nodes to CFG Nodes. This represents the size increase of PSCFG wrt CFG. The last column is the total time taken to produce the PSCFG. In general, the size increase ranges anywhere from 1.2 to 2 times the original CFG (that is, an increase of 20-100%). The geometric mean of the ratio is roughly 1.78 (an increase of 78%). The statemate program is an extreme case and deserves special mention as its PSCFG is comparatively large and takes much longer time to generate. This is mainly caused by the huge number of infeasible paths which is due to the fact that the program is synthesized automatically by the STAtechart tool. As a result, the node-splitting rule is triggered very frequently and hence, the PSCFG grows more significantly. Finally, we observed that the average time to build the PSCFG is around 18 seconds with a median of 4.1 seconds.

5.2 Effectiveness of PSCFG with Several Analyses

Our main second experiment aims at demonstrating that PSCFG can often enhance program analyses by producing important accuracy gains. We use three widely used program analyses: live variable analysis, alias analysis, and static backward slicing.

Our experiments take either the decompiled program (e.g., live and alias) or directly a PSCFG (e.g., slicing) and run the analysis of interest on it. Subsequently, we run the same analysis on the original C program or CFG and finally, compare results. We would like to clarify that the PSCFG 'per se' is not of our interest. We use

Live Variable Analysis		
Benchmark	P1	P2
cdaudio	6.0%	46.2%
fcron	9.1%	10.9%
floppy	13.5%	8.7%
kbfiltr	9.7%	21.9%
statemate	66.9%	1.8%
stunnel	13.2%	49.3%
tlan	4.2%	16.7%
Weighted Avg.	**12.4%**	
Geom. Mean		**14.6%**

(a)

Alias Analysis			
Benchmark	Original	Decompiled	Ratio
cdaudio	1833312	1722451	0.94
fcron	526158	477261	0.91
floppy	1640510	1523693	0.93
kbfiltr	18629	15747	0.85
pfinger	625325	569329	0.91
tlan	588252	556182	0.95
Geom. Mean			**0.91**
Improv.			**9%**

(b)

Backward Slicing			
Benchmark	N1	N2	Diff
cdaudio	1968	1227	741
diskperf	514	325	189
fcron	2594	1744	850
floppy	1794	1486	308
mpeg	1389	1331	58
serial	1616	1325	292
Weighted Avg.			**16.6%**

(c)

Figure 5. Effectiveness of PSCFG Using Live Variable Analysis, Alias Analysis and Backward Slicing

it in order for us to run the analysis on it and then, infer (hopefully) more precise information which can be applied to the original CFG for optimization or any other purpose. That is, we do not intend a PSCFG to be used in replacement of the original CFG. Therefore, for all the experiments we first describe how to "translate" the analysis information inferred on a PSCFG in order to be applied on the original CFG and then, we compare with the information inferred by running the analysis directly on the CFG.

Finally, we do not include the timing of running the off-the-shelf analyses on the PSCFG since the numbers are negligible compared with the construction time.

Live Variable Analysis. Live variable analysis [1] is a backward dataflow analysis that calculates for each program location the set of variables that may be potentially used before their next definition. Each variable of this set is called a *live* variable.

For comparison, we consider the number of live variables at each program location. Note that due to node splitting performed by our program transformation, each program location in the original program could potentially have multiple *instances* in the decompiled program. Hence we define the following rule in order to measure improvement at a program location. Let $L(\ell)$ be the set of live variables at program location ℓ. Then, we run the live variable analysis on the original and decompiled program. In the original program, $L(\ell)$ is computed by extracting directly the results of the analysis at ℓ. From the decompiled program, since there are multiple instances of ℓ: $\ell_1,\ldots,\ell_n$, we extract first the live variables for each instance $L(\ell_i)$ $(1 \le i \le n)$ and then, perform the union of all such sets (i.e.,$L(\ell) = \bigcup_{1\le i\le n} L(\ell_i)$).

Fig 5(a) shows the results of the live variable analysis using the analysis tool CIL [20]. The second column P1 shows the percentage of program points in which the decompiled program produced a more precise (smaller) set of live variables. The third column P2 shows, for those program points, the percentage of reduction using the decompiled program wrt the original. The percentage of reduction is obtained from the formula $\frac{n-m}{n} \times 100$ where n and m are the number of live variables in the program point in the original and decompiled programs respectively. In summary, our evaluation demonstrates that inferring live variable information on our decompiled program can remove 14.6% more live variables from 12.4% of the program points.

Alias Analysis. Alias analysis is concerned with whether two or more variables may point to the same memory location [1]. Two variables are said to be *aliased* to each other if they point to the same location. It is usually computed in a flow insensitive manner. That is, the order of statements in the program (the *flow*) is not considered. If two variables are aliased at some point in the program, then they are said to be in the same *equivalence class*.

We used the analysis tool Crystal [21] for this experiment. The concrete semantics of an equivalence class in Crystal is a set of the Cartesian product of all its elements. That is, if E is an equivalence class and $|E|$ its cardinality, then the total number of possible alias relationships in E is $\frac{|E|\times(|E|-1)}{2}$. This is the measure used to compare the effectiveness of the original CFG wrt to PSCFG. In Fig 5(b), we present the results for alias analysis. The second and third columns, Original and Decompiled, represent the total number of possible alias relationships for the original and decompiled programs, respectively. The ratio of Decompiled to Original is shown in the last column. The geometric mean of the ratio is 0.91, which concludes that the decompiled program shows, on average, an improvement of 9% (i.e., there are 9% less alias relationships).

It is worth mentioning that although the accuracy gains obtained can be considered sufficiently relevant, the flow-insensitive nature of the analysis limits our PSCFG to produce more precise results. A flow-insensitive analysis can benefit only from the removal of infeasible paths but not from the splitting of nodes, thus reducing the amount of precision that can be gained from the use of PSCFG.

Static Backward Slicing. The backward slice [27] of a program wrt to a program point ℓ and a set of variables $\mathcal{V}$ is defined as all statements of the program that might affect the value $\mathcal{V}$ at ℓ. As we illustrated at the end of Sec. 4, slicing relies on the computation of *postdominators* which are not preserved by PSCFG. Due to this limitation, we cannot decompile the PSCFG and feed it directly into an off-the-shelf slicer. Instead, we have implemented a static backward slicing following [27][12] which works directly on a PSCFG. The only modification done is that we perform all postdominator queries on the original CFG and then we map back the results to our PSCFG. For space reasons, we omit the details here because although trivial are tedious. We measure the improvement provided by PSCFG as follows: a transition in the original CFG is included in the slice if any of its *instances* in the PSCFG is included in the slice.

For the slicing criteria, we consider variables that may be of interest during debugging tasks. For the software model checking programs, we choose as slicing criteria the set of variables that appear in the safety conditions used for their verification in [13]. In the case of `mpeg` we choose a variable that contains the type of the video to be compressed. Finally, in `fcron` we choose all the file descriptors opened and closed by the application.

Fig 5(c) compares our slicer on the original CFG (column labelled with N1) against the same slicer on PSCFG (labelled with N2). Note that the measure used in both columns is the same, which is the number of nodes in the original CFG included in the slice: we map the results obtained from the PSCFG back to the CFG using

[12] Clearly, we could have implemented a more state-of-the-art program slicer (e.g., [14]). For simplicity, we decided to implement [27]. Nevertheless, we believe same conclusions apply, for instance, to [14].

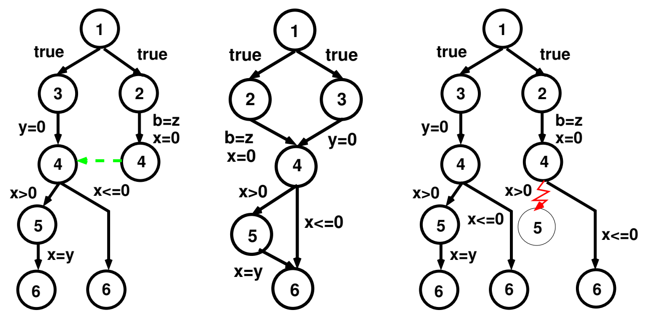

Figure 6. Subsumption May Hide Most Precise Results

the rule mentioned above. In the final column (Diff), we summarize the difference between the previous two columns.

The weighted average of the Diff column with respect to the number of nodes in each benchmark (obtained from Fig. 4) is about 16.6%. This shows that on an average, there are 16.6% less number of nodes in the slice of the CFG if the slicer is actually run on the PSCFG and has its results mapped back. The `mpeg` program is an exception since the number of lines sliced away in both columns is small compared to the other benchmarks. The reason is that in `mpeg` the size of the slices themselves is big because all computations in the program depend on the type of video to be compressed which is our slicing criteria.

Conclusions about Experiments of PSCFG. We have thus shown empirical evidence that PSCFG can be built in a reasonable amount of time with an affordable size of generally twice the size of the CFG using a set of real, medium-size programs. We have also demonstrated that the use of PSCFG generally pays off and it can improve in a range of 10% to 16% the results of three widely used program analyses: live variable analysis, alias analysis and backward slicing.

6. Backward Analysis with Witnesses

The soundness of our program transformation in Sec. 4 relies on the fact the set of traces of the PSCFG remains a superset of the exact traces of the original program. Moreover, the set of traces in the PSCFG is always a *subset* of the traces represented by the CFG. However, these conditions do not ensure that our transformation (PSCFG) produces the most precise results as the next example illustrates.

EXAMPLE 2. *Let us consider again the CFG in Fig. 1(c). Say symbolic execution explores first now the feasible path* $1 \to 3 \to 4 \to 5 \to 6$. *Then, we backtrack and explore* $4 \to 6$. *Since both paths arising from* 4 *are feasible the interpolant computed at* 4 *is true. As a consequence, the path* $1 \to 2 \to 4$ *can be subsumed. The interpolation-based symbolic tree is depicted in Fig. 6(a), and the transformed CFG is in Fig. 6(b). Note that the transformed CFG is equivalent to the original one. Therefore, a live variable analysis on it will not be able now to eliminate the variable y at* 1. □

This example has illustrated once again that the consideration of infeasible paths is essential in order to obtain the most precise results. Moreover, this example discovers that the search order of the symbolic execution matters due to the *asymmetric* behavior of subsumption. That is, the fact that a node is subsumed if its set of reachable paths is simply a subset, and not identical, of the reachable set of another node. Therefore, in order to avoid this non-deterministic behavior we need somehow to strengthen the condition that decides whether the symbolic execution can stop the

```
BackAna$_{\mathcal{A}}(\Pi)$
INPUT:  $\Pi$, a set of (possibly cyclic) sequences of transitions
OUTPUT: analysis, an array of locations with the analysis results
1:  foreach $\ell$ in nodes($\Pi$) do
2:      if $\ell \neq \ell_{end}$ then analysis[$\ell$] := $\bot$
3:      else analysis[$\ell$] := $\sigma_{init_{\mathcal{A}}}$
4:  end
5:  while no change in analysis do
6:      foreach $\ell \xrightarrow{op} \ell' \in \Pi$ do
7:          analysis[$\ell$] := analysis[$\ell$] $\bigsqcup_{\mathcal{A}} \widehat{pre}_{\mathcal{A}}$(analysis[$\ell'$], op)
8:      end
9:  end
10: return analysis
```

Figure 7. Backward Program Analysis [1]

exploration of a path and join symbolic states. In order to achieve this, we present the second component of our framework.

In this section, we present the second component of our framework which is a *generic* algorithm that interleaves the symbolic execution process described in Sec. 4 with the computation of the analysis so that the symbolic execution can use information from the analysis in order to decide better whether a symbolic state should be joined or not. The algorithm is generic in the sense that it is parameterized by the analysis. This means that in principle the only information provided by the designer is the analysis-specific operations (join operator, transfer function, ...). For efficiency reasons that we shall explain later, this algorithm only makes sense for backward analyses. Although more expensive, our preliminary results in Sec 6.2 indicate that this algorithm could be implemented efficiently and obtain some extra benefits.

We start by showing in Fig. 7 the classical algorithm to perform backward analysis on a set of (possibly cyclic) sequence of transitions by computing an iterative fixpoint algorithm [1]. Of course, the fact that we use as the input a set of paths makes sense if all these paths have a common starting point. The subscript $\mathcal{A}$ emphasizes the fact the algorithm is parameterized by the particular analysis $\mathcal{A}$. Now, we introduce the concept of *witness* that will be used to establish the conditions to ensure that a node can be joined without incurring in any loss of precision.

DEFINITION 3 (Witness Path). *Let P be a transition system and* Π *the set of reachable paths*[13] *from a state s in P. Given that* $a \in \mathsf{BackAna}_{\mathcal{A}}(\Pi)[s]$ *then we say* Ω_a *is a witness of the value a if:*

1 $\Omega_a \subseteq \Pi$ *and*

2 $a \in \mathsf{BackAna}_{\mathcal{A}}(\Omega_a)[s]$

Note hence that the notion of witness path Ω_a is associated with a particular state s (and its set of reachable paths Π) and with a particular lattice value a which is part of the analysis solution. Thus, each value of the solution may have a different witness path although in practice, they share many of the witness paths.

DEFINITION 4 (Minimal Witness Path). *We say* $\Omega_a \sqsubseteq \Omega'_a$ *if* $a \in \mathsf{BackAna}_{\mathcal{A}}(\Omega_a)[s] \cap \mathsf{BackAna}_{\mathcal{A}}(\Omega'_a)[s]$ *and* $\Omega_a \subseteq \Omega'_a$. *Then, a witness* Ω_a *is* minimal *if there does not exist another witness* Ω'_a *such that* $\Omega'_a \sqsubseteq \Omega_a$. *Note that a minimal witness always exists but it is not, in general, unique.*

Given a witness Ω we can construct a quantifier-free first-order logic formula, called *witness formula*, as follows. From each sequence of the set Ω, we built a conjunctive formula with all the

[13] Here, a path is simply a sequence of transitions. It should not be confused with a symbolic path which is a sequence of symbolic states.

constraints along the sequence. Then, the witness formula is the disjunction of each formula associated with each sequence. Note that the size of this formula can be, in general, exponential in the number of paths. In practice, we have observed that *not too many paths contribute to the analysis in different ways*. As a result, the size of the witness formulas is tractable.

EXAMPLE 3 (Witness Paths and Formulas). *Consider the CFG in Fig. 6(a). Let us focus on node* 4. *The set of live variables at* 4 *is* $\{x,y\}$. *A minimal witness path* Ω_x *for the variable x is either* $\{4{\to}5{\to}6\}$ *or* $\{4{\to}6\}$. *Note that the set* $\{4{\to}5{\to}6, 4{\to}6\}$ *is also a witness path but not minimal. It is easy to see that the backward analysis of any of these two paths will infer that x is live at* 4. *On the other hand, the witness path* Ω_y *for y can be only* $\{4{\to}5{\to}6\}$ *since y is only live when the path* $4{\to}5{\to}6$ *is analyzed. Assume that we choose* $\Omega_x \equiv \{4{\to}5{\to}6\}$ *as the witness for x. Then, its witness formula associated is simply the conjunction of the constraints along the path* $x > 0 \wedge x = y$. *Note that in this example y has the same witness path (and hence, same witness formula) than x.* □

We now present the overall process, described in Fig. 8, that takes advantage of the witness formulas in order to further improve the accuracy of the analysis enhanced by the use of PSCFG. The algorithm starts with an initial transition system P and symbolic state υ. The loop repeat in lines 3-19 executes the following steps until there is no change:

1 PSTransf generates $\mathcal{P}^+$ (i.e., a PSCFG) from a given state.

2 The foreach loop (lines 5-9) computes for every node in $\mathcal{P}^+$:
   (a) its analysis solution via $\mathsf{BackAna}_{\mathcal{A}}$ (line 7), and
   (b) set of minimal witness formulas for each value of the analysis solution (line 8) at the given node

3 The foreach loop (lines 10-17) identifies each node ℓ in $\mathcal{P}^+$ which was joined to another node ℓ'. For termination, we consider only *sibling-to-sibling* joins between nodes. A sibling-to-sibling join is when the two joined nodes refer to the same program location and they are connected through a back-edge[14]. Then, the set of witness formulas $\chi[\ell']$ is used as follows:

   For every node ℓ, we assume that we can obtain a formula Φ_ℓ that represents the symbolic state at the time ℓ was joined during the execution of PSTransf. This is easy to obtain by modifying PSTransf to annotate each node of the output transition system with that information. For any witness formula $\phi \in \chi[\ell']$ if the conjunction $\Phi_\ell \wedge \phi$ (so called *reuse condition*) is unsatisfiable (line 12) then it means that some feasible path reachable from ℓ' from which one analysis value was inferred, is now infeasible from ℓ. As a result, if ℓ would *reuse* the analysis solution from ℓ' directly, we may incur in some loss of precision. Therefore, if any of the reuse conditions is unsatisfiable, we refine $\mathcal{P}^+$ by unfolding the node ℓ and goto 1.

It is important to notice that the fact we decide to split more nodes from a given PSCFG using witness formulas does not affect the soundness of the analysis but only its accuracy. A discussion about the termination of $\mathsf{BackAnaWitness}_{\mathcal{A}}$ is necessary. At each iteration it is straightforward to see that the two foreach loops terminate since the number of nodes of a given PSCFG is always finite. The termination of the repeat loop is ensured because it is executed until either:

1 we reach a fixpoint, and hence, the reuse conditions are satisfied by all joined nodes, or

2 there are no more joined nodes which means that we did split all nodes. The number of splits is finite, exponentially bounded by the number of branches of the program. Note that we only split sibling-to-sibling joined nodes, and hence, we cannot unwind loops.[15]

[14] An edge from node u to v is called back-edge if v dominates u.

```
BackAnaWitness_A(ℓ, P)
INPUT:   υ, initial symbolic state
         P, transition system
OUTPUT: analysis, array of locations with the analysis results

1:  P⁺ := P, υ_init := υ
2:  change := false
3:  repeat
4:    P⁺ := PSTransf(υ_init, P⁺)
5:    foreach ℓ ∈ nodes(P⁺) do
6:      let Π_ℓ be the set of paths reachable from ℓ in P⁺
7:      analysis := BackAna_A(Π_ℓ)
8:      χ[l] := {φ_a | a ∈ analysis[l], φ_a is a minimal witness formula of a}
9:    endfor
10:   foreach ℓ ∈ nodes(P⁺) s.t.
              ℓ is sibling-to-sibling subsumed by ℓ' do
11:     let Φ_ℓ be the formula of the symbolic state at ℓ
12:     if ∃φ ∈ χ[ℓ'] • Φ_ℓ ∧ φ is unsatisfiable then
13:       υ_init := ⟨ℓ, s_Φ, Π_Φ⟩  /* s_Φ and Π_Φ are built from Φ */
14:       change := true
15:       goto L
16:     endif
17:   endfor
18:   L:
19: until not change
20: return analysis
```

Figure 8. Backward Program Analysis with Witnesses

EXAMPLE 4 (Use of Witnesses To Improve Precision). *Suppose we execute the algorithm* $\mathsf{BackAnaWitness}_{\mathcal{A}}$ *on the CFG in Fig. 1(c).*

In the first iteration of the repeat *loop, after line 4 we obtain the PSCFG of Fig. 6(a). The* foreach *loop (lines 5-9) computes the analysis values for each node and builds their corresponding witness formulas. Let us focus again on node* 4. *Recall that in Example 3 we computed the set of witness formulas* $\{\phi_x \equiv \phi_y \equiv x > 0 \wedge x = y\}$ *for the set of live variables* $\{x,y\}$. *The* foreach *loop (lines 10-17) identifies all points where we joined nodes during the construction of PSCFG. We notice that the path* $1{\to}2{\to}4$ *was joined with the other paths that pass through* $1{\to}3{\to}4$. *Then, we test if* $\exists \phi \in \{\phi_x, \phi_y\} \bullet \phi \wedge (x = 0 \wedge b = z)$ *is unsatisfiable. The formula* $x = 0 \wedge b = z$ *is obtained from the symbolic execution of the path* $1{\to}2{\to}4$. *Note what happens now. This test succeeds since* $\phi_x \wedge x = 0 \wedge b = z$ *is unsatisfiable. Then, we refine the PSCFG and continue with the execution of the path* $1{\to}3{\to}4$ *forcing a node-splitting at* 4. *In the second iteration of the* repeat *loop, we obtain the PSCFG of Fig 6(c) which cannot be further modified, producing the most precise answer (i.e., the live set* $\{x,z\}$ *at* 1*).* □

6.1 Implementation

The algorithm $\mathsf{BackAnaWitness}_{\mathcal{A}}$ is non-constructive in the sense that it does not explain how the witnesses formulas can be built more efficiently. Moreover, it is very inefficient since whenever a new node-splitting is done the whole process is repeated. To fill these gaps, we outline now a more realistic implementation of this algorithm. The central idea consists of augmenting the symbolic execution process performed by function IntpSymExec in Fig. 2 for computing the analysis answers using $\mathsf{BackAna}_{\mathcal{A}}$ in Fig. 7 on the symbolic tree while synthesizing its witness formulas which in

[15] For some specific analysis (e.g.,[26]) we allow unrolling loops a finite number of times.

	PSCFG+Slicing	BackAnaWitness$_{\mathrm{SLICING}}$
Benchmark	T(s)	T(s)
mpeg	60	628
diskperf	14	94
floppy	90	263
cdaudio	35	301
serial	92	395
fcron	133	832
Average	**70 sec**	**418 sec**

Figure 9. A Backward Slicing on PSCFG vs BackAnaWitness$_{\mathrm{SLICING}}$

turn are used by the symbolic execution in order to decide whether to join nodes or not.

Recall IntpSymExec returns for a given initial symbolic state υ its interpolant $\overline{\Psi}$ and the set of edges $\mathcal{E}$ that represent the symbolic execution tree built so far. We modify IntpSymExec to return the analysis answer $\sigma \in \mathcal{L}$ as well and its corresponding set of witness formulas $\chi \in FO$. We also modify the format of each entry in the subsumption table in order to store for each state υ the analysis answer σ and formulas χ. The interpolants and the set $\mathcal{E}$ are computed exactly as before. We now describe how to compute the two new output arguments of the algorithm. As we did before, let us omit the discussion about loops for now.

We have three base cases for each kind of leaf node in the tree. If an infeasible path is encountered we return $\perp$ and its set of witness formulas is $\emptyset$. If the path is feasible we return $\sigma_{init_{\mathcal{A}}}$ and its set $\{\langle v, true\rangle \mid v \in \sigma_{init_{\mathcal{A}}}\}$. For convenience, we represent a witness formula as a pair of a lattice element $v \in \mathcal{A}$ and its corresponding witness formula ϕ. The key step is during the subsumption test. Before, we simply applied Def. 2 in order to decide if the symbolic execution could halt and hence, join nodes. Now, we redefine Def. 2 for making use of witness formulas.

DEFINITION 5 (Subsumption with Interpolant and Witness Formulas). *Given two symbolic states υ and υ' such that υ is annotated with the interpolant $\overline{\Psi}$ and its set of witness formulas is χ. We say that υ' is* subsumed *by υ if:*

1 $[\![\upsilon']\!]$ implies $\overline{\Psi}$ (i.e., s.t. $[\![\upsilon']\!] \models \overline{\Psi}$) (as before)

2 $\forall \langle v, \phi\rangle \in \chi \bullet [\![\upsilon']\!] \wedge \phi$ is satisfiable

If the two conditions of Def. 5 hold, then the symbolic execution stops and the nodes are joined. In addition, we return the answer and witnesses from the subsumer node (υ).

Finally, we explain the recursive case. The forward symbolic execution remains the same. During the backtracking, we compute the answer σ from the children's answers σ' using the classical equation:

$$\sigma = \bigsqcup_{\sigma'} \widehat{pre}_{\mathcal{A}}(\sigma', \mathsf{op}) \tag{2}$$

A new step is to produce the witness formulas associated with σ. Assume recursively we have already computed χ' (the witness formulas of σ'). Then

$$\chi = \bigotimes_{\chi'} \{\langle v, \phi \wedge [\![\mathsf{op}]\!]\rangle \mid \langle v, \phi\rangle \in \chi'\} \tag{3}$$

where $\bigotimes$ is the join operator of witness formulas defined as:

$$\begin{aligned}\bigotimes Ss = \{&\langle v, \phi_1 \vee \phi_2\rangle \mid \langle v, \phi_1\rangle \in S, \langle v, \phi_2\rangle \in S\} \cup \\ \{&\langle v, \phi_1\rangle \mid \langle v, \phi_1\rangle \in S \text{ s.t. } \nexists \langle v, \phi_2\rangle \in S\}\end{aligned}$$

where $S = \bigsqcup Ss$. Thus, the join of witness formulas is quite straightforward. First, it applies the set union $\bigsqcup$. Then, whenever there are two candidate witness formulas (e.g., ϕ_1 and ϕ_2) for the same value v, we join them by introducing a disjunction.

In our current prototype we limit the number of disjunctions to keep smaller the witness formulas. Note that this limitation is always sound and more importantly, it does not affect the accuracy of the analysis. However, it may hamper subsumption (i.e., we may introduce unnecessary node-splitting). Certainly, this is a topic for future research. One clear possibility would be to represent the set of witness formulas via *Binary Decision Diagrams* (*BDDs*). By doing so, we may benefit from sharing which we have already observed is very common and we may also have an efficient way of handling disjunctive formulas without any limit.

We now discuss how we handle loops. Since we are interleaving the symbolic execution process and the execution of the analysis, whenever we have loops the analysis side runs a fixpoint computation. In practice, it is very important to avoid redundant work being *incremental*. That is, paths for which their analysis answers have not changed from one fixpoint iteration to another should not be re-executed by the symbolic execution.

Regarding witness formulas there are two main scenarios to consider. The first one, when the symbolic execution encounters a back-edge. Assuming that we have executed and analyzed all the loop exits first then we can simply return the set of witness formulas built from them. The other case is during a fixpoint computation which is a bit more elaborated. In the first iteration of fixpoint, the set of values are computed using Eq. 2 while their witness formulas are updated using Eq. 3. If the analysis stabilizes then, we return those formulas. Otherwise, we start a new iteration. The main idea is to respect all the witness formulas computed before and only add new ones if new values are added. It is also important to notice that witness formulas must respect the loop invariants. That is, in Eq. 3 a witness formula is updated by conjoining ϕ, the witness formula of its descendant, with the formula associated to $[\![\mathsf{op}]\!]$. Within a loop the formula $[\![\mathsf{op}]\!]$ is abstracted to *true*if it is not invariant through the loop. This suffices to produce invariant witness formulas.

Remark. Finally, we explain why our analysis algorithm with witnesses in Sec. 6 is only suitable for backward analyses in spite of that the description given in BackAnaWitness$_{\mathcal{A}}$ can take also, in principle, forward ones. The central idea for a sensible implementation relies on the fact that the exploration of a path can be avoided if some knowledge (i.e., reuse conditions) about the future is known. The only way to know about the future is to explore other paths, compute their analysis answers together with their witness formulas and then use them to prune other paths. This process is inherently backward, and in fact, it can be modeled in a elegant way using a dynamic programming setting as in [15].

6.2 Preliminary Results

We extended the prototype built for the experiments in Sec. 5 in order to implement the main ideas of BackAnaWitness$_{\mathcal{A}}$ following the implementation details described in Sec. 6.1. The main objective of these experiments was to demonstrate that a sensible implementation is plausible and that, under certain conditions, the use of witnesses can produce more accurate results than using only PSCFGs.

For our first experiment, we augmented the backward slicer that we implemented in Sec. 5 with witnesses formulas (called BackAnaWitness$_{\mathrm{SLICING}}$). Since the lattice of slicing is 2^{Vars} we keep for each variable in the answer its corresponding witness formula. Therefore, for a given node in the symbolic execution tree the number of witnesses tracked is bounded by the number of program variables. The main result of this experiment is that using witnesses we could not improve the precision provided by the PSCFGs. Analysis times are shown in Fig. 9 to illustrate the extra cost of building and testing witness formulas. Nevertheless, this may indicate that although the accuracy provided by a PSCFG is hard to predict, in general, due to the *asymmetric* behavior of subsumption, in practice, it may not be the case.

We then implemented a second analysis with witnesses BackAnaWitness$_{\mathrm{WCET}}$, which is a simplified version of *Worst-Case Ex-*

	PSCFG+WCET			BackAnaWitness$_{\text{WCET}}$			
Benchmark	B	N	T(s)	B	N	T(s)	W
`cdaudio`	443	5657	10	443	5657	13	0
`diskperf`	483	3931	8.4	483	3931	11	0
`kbfiltr`	154	1239	1.6	154	1239	2.4	0
`floppy`	457	2804	7	457	2804	9	0
`mpeg`	802	3927	6.4	802	4634	10.8	463
`qpmouse`	392	1018	1	392	1490	5	820
`statemate`	$\underline{265}$	53102	254	$\underline{261}$	94134	1021	10^5
`tlan`	$\underline{737}$	3917	5	$\underline{729}$	10547	42.2	76

Figure 10. A WCET Analysis on PSCFG vs BackAnaWitness$_{\text{WCET}}$

ecution Time (WCET) analysis. WCET analysis aims to compute the worst possible execution time of a program. Here, we focus on *high-level* analysis which aims at characterizing all possible executable paths excluding hardware effects. We implement a very simple timing model: programs are instrumented with a dedicated timing variable which is incremented after the execution of each statement. As usual, upon encountering a loop, it separately computes the WCET of the loop body and multiplies that value with a manually given number of loop iterations. The lattice members are of the form $Vars \mapsto Nat^{\top}$ where $\top$ is an infinite bound and $\sigma_{init} = 0$. The transfer function is $\widehat{pre}(\sigma', \mathsf{op}) = \{t + \mathsf{cost}(op) \mid t \in \sigma'\}$ where cost returns a positive number. The join operator is the maximum function between two positive numbers. Witness formulas are combined by selecting the witness associated with the maximum bound. Note that for WCET we need to keep track only of one witness formula corresponding to the dedicated timing variable at each node of the tree.

The results are shown in Fig. 10. Columns labelled by B show the constant upper bounds inferred (WCET). The columns N represent the number of nodes in PSCFG and in the case of BackAnaWitness$_{\text{WCET}}$, the number of nodes in the symbolic execution tree. Columns T(s) shows the time for constructing the PSCFG (the time for running the WCET analysis on PSCFG is not included since it is on the magnitude of milliseconds) and running BackAnaWitness$_{\text{WCET}}$. Finally, the column W is the number of times a witness formula was not preserved (i.e., Cond 2, Def. 5 did not hold), and hence, node splitting was performed.

Interestingly, there are two programs `statemate` and `tlan` where the use of witnesses improved the bound inferred (i.e., the upper bound was lower) compared with running the WCET analysis on the PSCFG. With the exception of `statemate` the analysis times running BackAnaWitness$_{\text{WCET}}$ show very little overhead wrt running the WCET analysis on the PSCFG. The reason why is the existence of only one witness formula at a given node.

7. Conclusions

We presented a framework for path-sensitive analysis, in two parts. The first and main part is a transformation for systematically converting a CFG into its path-sensitive counterpart PSCFG. The essential idea is that PSCFG exhibits fewer execution paths than the original CFG, and therefore produces more accuracy when subject to any analysis. Experiments then showed that the critical measure of size blowup from CFG to PSCFG is in fact modest, for medium-size programs. They also confirmed the folklore that considering path-sensitivity in program analysis generally produces significantly better results.

In the second part of this paper, we presented a generic backward algorithm that interleaves the construction of the PSCFG with the execution of the analysis. It uses the concept of *witnesses* which could now be augmented to the analysis so that subsumption is now conditional on the witnesses. Although more expensive this algorithm can produce more accurate results than the transformation of the first part. It is also not yet clear how much the witness concept helps across a range of analyses. But certainly it is crucial for analysis demanding a very high level of accuracy (e.g., WCET).

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *PLDI '98*, pages 72–84, 1998.

[3] R. Bodík, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *PLDI '97*, pages 146–158.

[4] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *FSE'97*, pages 361–377.

[5] R. Bodíkv and S. Anik. Path-sensitive value-flow analysis. In *POPL '98*, pages 237–251, 1998.

[6] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *4th POPL*, pages 238–252. ACM Press, 1977.

[7] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL'78*.

[8] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Computation*, 22, 1955.

[9] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68, 2002.

[10] Dinakar Dhurjati, Manuvir Das, and Yue Yang. Path-Sensitive Dataflow Analysis with Iterative Refinement. In *Static Analysis Symposium*, 2006.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.

[12] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *ESEC/FSE-13*, pages 227–236, 2005.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88*, pages 35–46.

[15] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAAI*, pages 297–303. AAAI Press, 2008.

[16] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *15th CP*, volume 5732 of *LNCS*. Springer, 2009.

[17] James C. King. Symbolic Execution and Program Testing. *Com. ACM' 76*, pages 385–394.

[18] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05*, pages 5–20, 2005.

[19] K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC'02*.

[21] Radu Rugina, Maksim Orlovich, and Xin Zheng. Crystal: A program analysis system for C. http://www.cs.cornell.edu/projects/crystal, 2007. [Online; accessed 09-July-2011].

[22] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. volume 15, pages 410–457, 2006.

[23] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC '06*, 2006.

[24] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *APLAS'03*, pages 230–245.

[25] A. Thakur and R. Govindarajan. Comprehensive path-sensitive dataflow analysis. In *CGO '08*, pages 55–63, 2008.

[26] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Syst.*, 18:157–179, 2000.

[27] M. Weiser. Program slicing. In *ICSE '81*, pages 439–449, 1981.