Path-Sensitive Backward Slicing

Joxan Jaffar[1], Vijayaraghavan Murali[1], Jorge A. Navas[2], and Andrew E. Santosa[3]

[1] National University of Singapore
[2] The University of Melbourne
[3] University of Sydney

Abstract. Backward slicers are typically path-insensitive (i.e., they ignore the evaluation of predicates at conditional branches) often producing too big slices. Though the effect of path-sensitivity is always desirable, the major challenge is that there are, in general, an exponential number of predicates to be considered. We present a *path-sensitive* backward slicer and demonstrate its practicality with real C programs. The crux of our method is a symbolic execution-based algorithm that excludes spurious dependencies lying on infeasible paths and avoids imprecise joins at merging points while reusing dependencies already computed by other paths, thus pruning the search space significantly.

1 Introduction

Weiser [19] defined the *backward slice* of a program with respect to a program location ℓ and a variable x, called the slicing criterion, as all statements of the program that might affect the value of x at ℓ, considering all possible executions of the program. Slicing was first developed to facilitate software debugging, but it has subsequently been used for performing diverse tasks such as parallelization, software testing and maintenance, program comprehension, reverse engineering, program integration and differencing, and compiler tuning.

Although static slicing has been successfully used in many software engineering applications, slices may be quite imprecise in practice - "*slices are bigger than expected and sometimes too big to be useful [2]*". Two possible sources of imprecision are: inclusion of dependencies originated from *infeasible paths*, and merging abstract states (via join operator) along incoming edges of a *control flow merge*. A systematic way to avoid these inaccuracies is to perform path-sensitive analysis. An analysis is said to be *path-sensitive* if it keeps track of different state values based on the evaluation of the predicates at conditional branches. Although path-sensitive analyses are more precise than both flow-sensitive and context-sensitive analyses they are very rare due to the difficulty of designing efficient algorithms that can handle its combinatorial nature.

The main result of this paper is a practical path-sensitive algorithm to compute backward slices. *Symbolic execution (SE)* is the underlying technique that provides path-sensitiveness to our method. SE uses symbolic inputs rather than actual data and executes the program considering those symbolic inputs. During the execution of a path all its constraints are accumulated in a formula P. Whenever code of the form if(C) then S1 else S2 is reached the execution forks the current state and updates the two copies $P_1 \equiv P \wedge C$ and $P_2 \equiv P \wedge \neg C$, respectively. Then, it checks if either P_1 or P_2 is unsatisfiable. If yes, then the path is *infeasible* and hence, the execution stops and backtracks to the last choice point. Otherwise, the execution continues. The set of all paths explored by symbolic execution is called the *symbolic execution tree (SET)*.

Not surprisingly, a backward slicer can be easily adapted to compute slices on SETs rather than control flow graphs (CFGs) and then mapping the results from the SET to the original CFG. It is not difficult to see that the result would be a fully path-sensitive slicer. However, there are two challenges facing this idea. First, the *path explosion problem* in path-sensitive analyses that is also present in SE since the size of the SET is exponential in the number of conditional branches. The second challenge is the infinite length of symbolic paths due to unbounded loops. To overcome the latter we borrow from [17] the use of inductive invariants produced from an abstract interpreter to automatically compute *approximate loop invariants*. Because invariants are approximate our algorithm cannot be considered fully path-sensitive in the presence of loops. Nevertheless our results in Sec. 5 demonstrate that our approach can still produce significantly more precise slices than a path-insensitive slicer.

Therefore, the main technical contribution of this paper is how to tackle the path-explosion problem. We rely on the observation that *many symbolic paths have the same impact on the slicing criterion*. In other words, there is no need to explore all possible paths to produce the most precise slice. Our method takes advantage of this observation and explores the search space by dividing the problem into smaller sub-problems which are then solved recursively. Then, it is common for many sub-problems to be *"equivalent"* to others. When this is the case, those sub-problems can be skipped and the search space can be significantly reduced with exponential speedups. In order to successfully implement this search strategy we need to (a) store the solution of a sub-problem as well as the conditions that must hold for reusing that solution, (b) reuse a stored solution if a new encountered sub-problem is "equivalent" to one already solved.

Our approach symbolically executes the program in a depth-first search manner. This allows us to define a sub-problem as any subtree contained in the SET. Given a subtree, our method following Weiser's algorithm computes dependencies among variables that allow us to also infer which statements may affect the slicing criterion. The fundamental idea for reusing a solution is that when the set of feasible paths in a given subtree is *identical* to that of an already explored subtree, it is not possible to deduce more accurate dependencies from the given subtree. In such cases we can safely reuse dependencies from the explored subtree. However, this check is impractical because it is tantamount to actually exploring the given subtree, which defeats the purpose of reuse. Hence we define certain reusing conditions, the cornerstone of our algorithm, which are both sound and precise enough to allow reuse without exploring the given subtree.

First, we store a formula that succinctly captures all the infeasible paths detected during the symbolic execution of a subtree. We use efficient *interpolation* techniques to generate *interpolants* [5] for this purpose. Then, whenever a new subtree is encountered we check if the constraints accumulated *imply* in the logical sense the *interpolant* of an already solved subtree. If not, it means there are paths in the new subtree which were unexplored (infeasible) before, and so we need to explore the subtree in order to be sound. Otherwise, the set of paths in the new subtree is a *subset* of that of the explored subtree. However, being a subset is not sufficient for reuse since we need to know if they are *equivalent*, but the equivalence test, as mentioned before, is impractical. Here, we make use of our intuition that only few paths contribute to the dependency information in every subtree. Hence, to check for equivalence of subtrees we need not check all

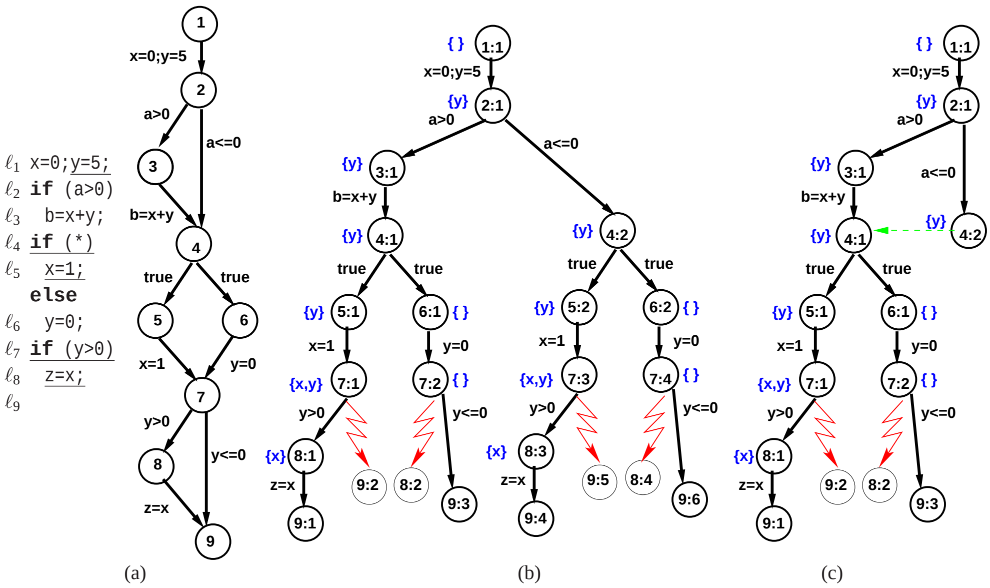

Fig. 1. (a) A program and its transition system, (b) its naive symbolic execution tree (SET) and (c) its interpolation-based SET, for slicing criterion $\langle \ell_9, \{z\} \rangle$. The final slice consists of the underlined statements.

paths, but only those few that contributed to the dependencies, what we call the *witness paths*. Now, if the previous implication succeeds we also check if the conjunctions of constraints along the witness paths of the explored subtree are satisfiable in the new subtree. If yes, we reuse dependencies. Otherwise, the equivalence test failed.

Finally, it is worth mentioning that some previous works have tackled the problem of path-sensitive backward slicing before as we will discuss them in Sec. 6. However, to the best of our knowledge either they suffer from the path-explosion problem or efficiency is achieved at the expense of losing some path-sensitiveness. One essential result of our method is that it produces *exact* slices for loop-free programs. By "exact" we mean that the algorithm guarantees to not produce dependencies from spurious[1] (i.e., non-executable) paths. In other words, it produces the *smallest* possible, *sound* slice of a loop-free program for any given slicing criterion.

2 Motivating Example

We first describe our approach through an example. Consider the program in Fig. 1(a) and assume we would like to slice it wrt location ℓ_9 and variable z. The assignment x=0 at ℓ_1 should not be included in the slice because any path that reaches ℓ_8 through ℓ_5 redefines x and any path that reaches ℓ_8 through ℓ_6 (without redefining x) is infeasible. Note that a path insensitive algorithm would not be able to infer this from the CFG.

Fig. 1(b) shows the naive symbolic execution tree of the program. The nodes are labeled with $\ell : k$ (ℓ is a program location and k is an identifier to distinguish nodes with

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

the same program location belonging to different symbolic paths) and edges between two locations are labeled by the intervening program operation. Solid (black) edges denote feasible transitions and zigzag (red) edges denote infeasible transitions. Each node is annotated with its *dependency set* between brackets (blue) obtained by running Weiser's [19] algorithm. Informally, a dependency set at location ℓ contains all variables that may affect the slicing criterion from any path reachable from ℓ. A statement at ℓ is included in the slice if the intersection between the dependency set and the set of variables both defined at ℓ (i.e., left-hand side of the assignment) is not empty. Note that the dependency set at 2:1 only contains y and therefore, the statement x=0 at ℓ_1 would not be included in the slice. Hence it is clear that the path-sensitive SET improves the accuracy of slices. The problem is that the size of the tree is exponential in the number of branches. However, consider now the tree[2] in Fig. 1(c) constructed by our method where dotted (green) edges denote reusing transitions. This tree contains the same relevant information needed to exclude x=0 from the slice but without some redundant paths present in Fig. 1(b). Let us see how our method generates the tree in Fig. 1(c).

Our algorithm performs symbolic execution guided by depth-first search exploring first the path $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_8 \cdot \ell_9$. As usual, it accumulates the constraints along the path in a formula Π, where variable redefinitions are denoted by primed versions. For the above path, $\Pi_{9:1} \equiv x = 0 \wedge y = 5 \wedge a > 0 \wedge b = x + y \wedge x' = 1 \wedge y > 0 \wedge z = x'$ is the formula built at 9:1, which is satisfiable. It then applies Weiser's algorithm to compute the dependency set at each node along the path. In addition, it also computes at each node one of the reusing conditions: the (smallest possible) set of paths from which the dependency set was generated. For example, at 7:1 the dependency set $\{x,y\}$ was obtained from the suffix path $\ell_7 \cdot \ell_8 \cdot \ell_9$, at 4:1 the dependency set $\{y\}$ was obtained from $\ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_8 \cdot \ell_9$, and so on. These paths are called the *witness paths* and they represent the paths along which each variable in the dependency set affects the slicing criterion.

Next our algorithm backtracks and explores the path $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_9$ with constraints $\Pi_{9:2} \equiv x = 0 \wedge y = 5 \wedge a > 0 \wedge b = x + y \wedge x' = 1 \wedge y \leq 0$. This formula is unsatisfiable and hence the path is infeasible. Now it generates another reusing condition: a formula called *interpolant* that captures the essence of the reason of infeasibility of the path. The main purpose of the interpolant is to exclude irrelevant facts pertaining to the infeasibility so that the reusing conditions are more likely to be reusable in future. For the above path a possible interpolant is $y = 5$ which is enough to capture its infeasibility and the infeasibility of any path that carries the constraint $y \leq 0$. In summary, our algorithm generates two kind of reusing conditions: witness paths from feasible paths and interpolants from infeasible paths.

Next it backtracks and explores the path $\ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4 \cdot \ell_6 \cdot \ell_7$. At 7:2, it checks whether it can reuse the solution from 7:1 by checking first if the accumulated constraints $\Pi_{7:2} \equiv x = 0 \wedge y = 5 \wedge a > 0 \wedge b = x + y \wedge y' = 0$ *imply* the interpolant at 7:1, $y' = 5$[3]. Since the implication fails, it has to explore 7:2 in order to be sound. The subtree after exploring this can be seen in Fig. 1(c). An important thing to note here is

[2] In fact, it is a Directed Acyclic Graph (DAG) due to the existence of reusing edges.

[3] The variable versions used in the interpolants must be properly renamed to be consistent with the versions used in a formula Π. For instance, here we know that the interpolant at 7:1 must be $y' = 5$, where y' is the newest version of y used in $\Pi_{7:2}$.

that while applying Weiser's algorithm, it has obtained a more accurate dependency set (empty set) at 7:2 than that which would have been obtained if it reused the solution from 7:1. Also note that at 4:1, the dependency set is still $\{y\}$ with witness path $\ell_4 \cdot \ell_5 \cdot \ell_7 \cdot \ell_8 \cdot \ell_9$ and interpolant $y = 5$.

Note what happens now. When our algorithm backtracks to explore the path $\ell_1 \cdot \ell_2 \cdot \ell_4$, it checks at 4:2 if it can reuse the solution from 4:1. This time, the accumulated constraints $x = 0 \wedge y = 5 \wedge a \leq 0$ imply the interpolant at 4:1, $y = 5$. In addition, the witness path at 4:1 is also feasible under 4:2. Hence, it simply reuses the dependency set $\{y\}$ from 4:1 both in a *sound* and *precise* manner, and backtracks without exploring 4:2. In this way, it prunes the search space while still maintaining as much as accuracy as the naive SET in Fig. 1(b). Now, when Weiser's algorithm propagates back the dependency set $\{y\}$ from 4:2, we get the dependency set $\{y\}$ again at 2:1, and the statement x=0 at 1:1 is not included in the slice.

3 Background

Syntax. We restrict our presentation to a simple imperative programming language 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* $\mathsf{x} = \mathsf{e}$ 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 then 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 location (with special symbols for initial location, ℓ_{start}, and final location, ℓ_{end}). The symbolic store s is a function from program variables to terms over input symbolic variables. Each program variable is initialized to a fresh input symbolic variable. The *evaluation* $[\![c]\!]_S$ of a constraint expression c in a store s is defined recursively as usual: $[\![v]\!]_S = s(v)$ (if $c \equiv v$ is a variable), $[\![n]\!]_S = n$ (if $c \equiv n$ is an integer), $[\![e \ \mathsf{op}_r \ e']\!]_S = [\![e]\!]_S \ \mathsf{op}_r \ [\![e']\!]_S$ (if $c \equiv e \ \mathsf{op}_r \ e'$ where e, e' are expressions and op_r is a relational operator $<, >, =, !=, >=, <=$), and $[\![e \ \mathsf{op}_a \ e']\!]_S = [\![e]\!]_S \ \mathsf{op}_a \ [\![e']\!]_S$ (if $c \equiv e \ \mathsf{op}_a \ e'$ where e, e' are expressions and op_a is an arithmetic operator $+, -, \times, \ldots$). 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 *FOL* 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:

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

Note that Eq. (1) queries a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound but not necessarily 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 $[\![\upsilon]\!] : \mathit{SymStates} \to \mathit{FOL}$ as the formula $(\bigwedge_{v \,\in\, \mathit{Vars}} [\![v]\!]_S) \wedge \Pi$ where *Vars* is the set of program 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 an *infeasible* state. 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* contains all the execution paths explored during the symbolic execution of a transition system by triggering Eq. (1). The nodes represent symbolic states and the arcs represent transitions between states.

Program Slicing. The *backward slice* of a program wrt a program location ℓ and a set of variables $V \subseteq \mathit{Vars}$, called the *slicing criterion* $\langle \ell, V \rangle$, is all statements of the program that might affect the values of V at ℓ.[4] We follow the dataflow approach described by Weiser [19] reformulated as an abstract domain $\mathcal{D} \equiv \{\bot\} \cup \mathcal{P}(\mathit{Vars})$ (where $\mathcal{P}(\mathit{Vars})$ is the powerset of program variables) with a lattice structure $\langle \sqsubseteq, \bot, \sqcup, \sqcap, \top \rangle$, such that $\sqsubseteq \equiv \subseteq$, $\sqcup \equiv \cup$, and $\sqcap \equiv \cap$ are conveniently lifted to consider the element $\bot$.

We say $\sigma_\ell \in \mathcal{D}$ is the approximate set of variables at location ℓ that may affect the slicing criterion. We will abuse notation to denote the dependencies associated to a symbolic state υ also as σ_υ. *Backward data dependencies* can be formulated using this set, defining two kinds of dataflow information. Given a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$ we define *def*(op) and *use*(op) as the sets of variables altered and used during the execution of op, respectively. Then,

$$\sigma_\ell \triangleq \begin{cases} (\sigma_{\ell'} \setminus \mathit{def}(\mathsf{op})) \cup \mathit{use}(\mathsf{op}) & \text{if } \sigma_{\ell'} \cap \mathit{def}(\mathsf{op}) \neq \emptyset \\ \sigma_{\ell'} & \text{otherwise} \end{cases} \tag{2}$$

where $\sigma_{\ell'} = V$ if $\ell' = \ell_{\mathsf{end}}$. We say a transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$ where $\mathsf{op} \equiv \mathsf{x = e}$ is included in the slice if:

$$\sigma_{\ell'} \cap \mathit{def}(\mathsf{op}) \neq \emptyset \tag{3}$$

Backward control dependencies can also affect the slicing criterion. A transition relation $\delta \equiv \ell \xrightarrow{\mathsf{op}} \ell'$ where $\mathsf{op} \equiv \mathsf{assume(c)}$ is included in the slice if any transition under the range of influence of δ (any path between δ and its *nearest postdominator* [19] in the transition system) is included in the slice, and (4)

[4] W.l.o.g., we assume in this paper a single slicing criterion at ℓ_{end}.

- $\sqcup : \mathcal{D}^\omega \times \mathcal{D}^\omega \to \mathcal{D}^\omega$

$$\sigma^\omega{}_1 \sqcup \sigma^\omega{}_2 \triangleq \sigma^\omega{}_1 \cup \sigma^\omega{}_2$$

- $\sqsubseteq : \mathcal{D}^\omega \times \mathcal{D}^\omega \to Bool$

$$\sigma^\omega{}_1 \sqsubseteq \sigma^\omega{}_2 \text{ if and only if } \sigma^\omega{}_1 \subseteq \sigma^\omega{}_2$$

- $\widehat{pre} : \mathcal{D}^\omega \times (\Sigma \times \Sigma \times Ops) \times (Vars \to SymVars) \to \mathcal{D}^\omega$.

$$\widehat{pre}(\sigma^{\omega\prime}, \ell \xrightarrow{\mathsf{op}} \ell', s) \triangleq \begin{cases} \textbf{let } \sigma^\omega = \widehat{pre\_aux}(\sigma^{\omega\prime}, \ell \xrightarrow{\mathsf{op}} \ell', s) \\ \quad \textbf{foreach } \langle x, \omega_x \rangle \in \sigma^\omega, \langle x, \omega_{x'} \rangle \in \sigma^\omega \\ \quad\quad \sigma^\omega = \sigma^\omega \setminus \{\langle x, \omega_x \rangle, \langle x, \omega_{x'} \rangle\} \\ \quad\quad \textbf{if } \omega_x \models \omega_{x'} \textbf{ then } \sigma^\omega = \sigma^\omega \cup \{\langle x, \omega_{x'} \rangle\} \\ \quad\quad \textbf{else } \sigma^\omega = \sigma^\omega \cup \{\langle x, \omega_x \rangle\} \\ \quad \textbf{if } (\sigma^\omega \cap def(\mathsf{op}) \textbf{ or } \mathsf{INFL}(\ell \to \ell') \cap \mathbf{S} \neq \emptyset) \textbf{ then} \\ \quad\quad \mathbf{S} = \mathbf{S} \cup \{\ell \to \ell'\} \\ \textbf{in } \sigma^\omega \end{cases}$$

where:

$$\widehat{pre\_aux}(\sigma^{\omega\prime}, \ell \xrightarrow{\mathsf{op}} \ell', s) \triangleq \begin{cases} \{\langle x, \omega_x \wedge [\![y = e]\!]_S \rangle \mid \langle x, \omega_x \rangle \in \sigma^{\omega\prime}, \mathsf{op} \equiv \mathsf{y = e},\ x \notin def(\mathsf{op})\} \cup \\ \{\langle v, \omega_x \wedge [\![y = e]\!]_S \rangle \mid \langle x, \omega_x \rangle \in \sigma^{\omega\prime}, \mathsf{op} \equiv \mathsf{y = e},\ x \in def(\mathsf{op}), v \in use(\mathsf{op})\} \cup \\ \{\langle x, \omega_x \wedge [\![c]\!]_S \rangle \quad \mid \langle x, \omega_x \rangle \in \sigma^{\omega\prime}, \mathsf{op} \equiv \mathsf{assume(c)}\} \cup \\ \{\langle x, [\![\Pi_\pi]\!]_S \wedge [\![c]\!]_S \rangle \mid \langle x, \cdot \rangle \notin \sigma^{\omega\prime}, \mathsf{op} \equiv \mathsf{assume(c)}, x \in use(\mathsf{op}), \\ \qquad\qquad\qquad \mathsf{INFL}(\ell \to \ell') \cap \mathbf{S} \neq \emptyset, \exists\, \pi \equiv \ell' \cdot \ldots \cdot \ell_{end}\} \end{cases}$$

Fig. 2. Main Abstract Operations for $\mathcal{D}^\omega$

$$\sigma_\ell \triangleq \sigma_{\ell'} \cup use(\mathsf{op}) \tag{5}$$

Finally, a function $\widehat{pre}_{\mathcal{D}}(\sigma_\ell, \mathsf{op})$ that returns the *pre-state* after executing backwards the operation op with the *post-state* σ_ℓ is defined using Eqs. (2),(3),(4), and (5).

4 Algorithm

A path-sensitive slicing algorithm over a symbolic execution tree (SET) can be defined as an *annotation* process which labels each symbolic state $\upsilon \equiv \langle \ell, \cdot, \cdot \rangle$ with $\sigma_\ell \in \mathcal{D}$ by computing a fixpoint (later formalized) over the tree, using Eqs. (2) and (5) described in Sec. 3. In an interleaved process, the final SET is obtained through Eqs. (3) and (4). Since the SET may have multiple instances of the same transition relation, we say that a transition relation is included in the final slice if at least one of its instances is included in the slice on the SET. It is easy to see that the path-sensitiveness comes from how symbolic execution builds the tree since no dependencies from a non-executable path can be considered.

Our algorithm performs symbolic execution in a depth-first search manner excluding all infeasible paths. Whenever the forward traversal of a path finishes due to a (a) terminal state, (b) infeasible state, or (c) reusing state (i.e., a state reusing a solution from another state), the algorithm halts and backtracks to the next path. During this backtracking each symbolic state υ is labelled with its *solution*, i.e., the set of variables σ_υ at υ that may affect the slicing criterion. Furthermore, the reusing conditions are computed at each state for future use. We first introduce formally the two key concepts which will decide whether a solution can be reused or not.

Definition 1 (Interpolant). *Given two first order logic (FOL) formulas A and B such that $A \wedge B$ is $false$ a Craig* interpolant *[5] wrt A is another FOL formula $\overline{\Psi}$ such that (a) $A \models \overline{\Psi}$, (b) $\overline{\Psi} \wedge B$ is false, and (c) $\overline{\Psi}$ is formed using common variables of A and B.*

Interpolation allows us to remove irrelevant facts from A without affecting the unsatisfiability of $A \wedge B$. It is worth mentioning that efficient interpolation algorithms exist for quantifier-free fragments of theories such as linear real/integer arithmetic, uninterpreted functions, pointers and arrays (e.g., [4]) where interpolants can be extracted from the refutation proof in linear time on the size of the proof.

Definition 2 (Witness Paths and Formulas). *Given a symbolic state $\upsilon \equiv \langle \ell, \cdot, \cdot \rangle$ annotated with the set of variables σ_υ that affect the slicing criterion at ℓ_{end}, a* witness path *for a variable $v \in \sigma_\upsilon$ is a symbolic path $\pi \equiv \langle \ell, \cdot, \cdot \rangle \cdot \ldots \cdot \langle \ell_{end}, \cdot, \Pi_{end} \rangle$ with the final symbolic state $\upsilon' \equiv \langle \ell_{end}, \cdot, \Pi_{end} \rangle$ such that $[\![\upsilon']\!]$ is satisfiable (i.e., π is feasible). We call $[\![\upsilon']\!]$ the* witness formula *of v, denoted ω_v.*

Intuitively, a witness path for a variable at a node is a path below the node along which the variable affects the slicing criterion at the end. A witness formula represents a condition sufficient for the variable to affect the slicing criterion along the witness path.

Prior to establishing the reusing conditions, we augment the abstract domain $\mathcal{D}$ to accommodate the witness formulas. Here, and in the rest of the paper, we will refer to the term "dependency" as the set of variables that may affect the slicing criterion together with their witnesses.

Definition 3 ($\mathcal{D}^\omega$). *We define a new abstract domain $\mathcal{D}^\omega$ as a lattice $\langle \sqsubseteq, \perp, \sqcup, \top \rangle$ such that $\mathcal{D}^\omega \triangleq \{\perp\} \cup \mathcal{P}(\mathrm{Vars} \times \mathrm{FOL})$ (i.e., set of pairs of the form $\langle x, \omega_x \rangle$ where x is a variable and ω_x is its witness formula) and abstract operations described in Fig. 2.*[5]

Note that the witness formulas can be obtained only from (feasible) paths in the program. Therefore, the number of witness formulas is always finite. As we will see later, even with loops, the size of each witness formula is also finite because we make the symbolic subtree of the loop finite. That is, we perform symbolic execution on a finite program once loop invariants are given. This ensures that the abstract domain $\mathcal{D}^\omega$ is finite and hence, termination is guaranteed for any fixpoint computation based on it.

In Fig. 2, the operator $\sqcup$ computes the least upper bound of the abstract states by simply applying the set union of the two set of states. The operator $\sqsubseteq$ simply tests whether one set is a subset of the other. $\widehat{pre}$ is a bit more elaborated but basically consists of the Eqs. (2), (3), (4), and (5) defined in Sec. 3 extended with witnesses formulas. We assume here and in the algorithm in Fig. 3 that $\widehat{pre}$ accesses **S** which is the set of transitions included in the slice so far. In function $\widehat{pre}\_aux$, there are four cases to handle different kinds of statements and dependencies:

- In the first two cases, if the operation is an assignment, the dependencies are propagated from the *defined* to the *used* variables and any dependency from a variable not *defined* is kept. In these cases, the pre-state witness formula is the conjunction of the post-state witness formula with the corresponding statement.

[5] For lack of space, trivial treatment of the element $\perp$ is omitted from operations in Fig. 2.

- In the third case, if the operation is an assume, any *used* variable is preserved, with its pre-state witness formula being the conjunction of the post-state witness formula and the corresponding guard.
- In the last case, for any variable x occurring in an assume statement without any dependency, if any transition under the range of influence [19] (computed by INFL) of the assume is already in the slice, then x is added (due to control dependency) and its witness formula is the conjunction of the guard and the path condition of any (feasible) path from the assume statement that leads to the end of the program.

In addition, in function $\widehat{pre}$ whenever two pairs from the set of dependencies computed by $\widehat{pre}\_aux$ refer to the same variable, we use an entailment test to choose the one with the weaker witness formula (which is more likely to be reused). In practice, the entailment test can be skipped by choosing arbitrarily one. Finally, a transition is included in the slice if one of the Eqs. (3) and (4) holds.

Definition 4 (Reusing Conditions). *Given a current symbolic state $\upsilon \equiv \langle \ell, \cdot, \Pi \rangle$ and an already solved symbolic state $\upsilon' \equiv \langle \ell, \cdot, \cdot \rangle$ such that $\overline{\Psi}$ is the interpolant generated for υ' and σ^ω are the dependencies together with their attached witnesses at υ', we say υ is* equivalent *to υ' (or υ can* reuse *the solution at υ') if the following conditions hold:*

$$(a)\ \ [\![\upsilon]\!] \models \overline{\Psi} \qquad (b)\ \ \forall \langle x, \cdot \rangle \in \sigma^\omega \bullet \exists \langle x, \omega_x \rangle \in \sigma^\omega \text{ such that } [\![\upsilon]\!] \wedge \omega_x \text{ is satisfiable} \quad (6)$$

The condition (a) affects *soundness* and it ensures that the set of symbolic paths reachable from υ must be a subset of those from υ'. The condition (b) is the witness check which essentially states that for each variable x in the dependency set at υ', there must be at least one witness path with formula ω_x that is feasible from υ. This affects *accuracy* and ensures that the reuse of dependencies does not incur any loss of precision.

We now describe in detail the main features of our algorithm defined by the function $\mathsf{BackwardDeps}_V$ in Fig. 3. The main purpose of $\mathsf{BackwardDeps}_V$ is to keep track of the *backward dependencies* between the program variables and the slicing criterion by inferring for each state the set of variables that may affect the slicing criterion. From these dependencies it is straightforward to obtain the slice of the program as explained at the beginning of this section. For clarity of presentation, let us omit for now the content of the *grey* boxes and assume programs do not have loops, which we will come to later.

$\mathsf{BackwardDeps}_V$: $SymStates \times \mathcal{D}^\omega \to FOL \times \mathcal{D}^\omega \times Bool$ requires the program to have been translated to a transition system $\langle \Sigma, I, \longrightarrow, O \rangle$ and taking an initial symbolic state $\upsilon \equiv \langle \ell \in I, \varepsilon, true \rangle$ and an initially empty σ^ω. V is the set of variables of the slicing criterion. The set of transitions included in the slice, **S**, is also empty. Recall that **S** is only modified by $\widehat{pre}_{\mathcal{D}^\omega}$, and hence, we omit it from the description of the algorithm defining it as a global variable. The returned value is a triple with the interpolant, dependencies (i.e., reusing conditions and solution) and a boolean flag representing whether any change occurred in a dependency set at any symbolic state during the algorithm's backward traversal (this is used mainly to handle loops later). The actual object of interest computed by the algorithm is the set of transitions **S** included in the slice.

$\mathsf{BackwardDeps}_V$ implements a recursive algorithm whose objective is to generate a finite complete SET while reusing solutions whenever possible to avoid path explosion.

$\mathsf{BackwardDeps}_V(\upsilon \equiv \langle \ell, s, \Pi \rangle, \sigma^{\omega})$
1: $\mathsf{change} = \mathit{false}$
2: **if** $\mathsf{INFEASIBLE}(\upsilon)$ **then** $\langle \overline{\Psi}, \sigma^{\omega} \rangle = \langle \mathit{false}, \emptyset \rangle$ and **goto** 12
3: **if** $\mathsf{TERMINAL}(\upsilon)$ **then** $\langle \overline{\Psi}, \sigma^{\omega} \rangle = \langle \mathit{true}, \{\langle v, \mathit{true} \rangle \mid v \in V\} \rangle$ and **goto** 12
4: **if** $\exists\, \upsilon' \equiv \langle \ell, s, \cdot \rangle$ labelled with $\langle \overline{\Psi}, \sigma^{\omega} \rangle$ such that $\mathsf{REUSE}(\upsilon, \upsilon')$ **then goto** 12
5: **if** ℓ is the header of a loop **then**
6: $\quad \overline{\upsilon} = \mathsf{invariant}(\upsilon, \ell \to \ldots \to \ell)$
7: $\quad \langle \overline{\Psi}, \sigma^{\omega}, \mathsf{change} \rangle = \mathsf{UnwindTree}_V(\overline{\upsilon}, \sigma^{\omega})$ and **goto** 12
8: **if** $\exists\, \ell'$ such that $\ell \to \ell'$ is a backedge of a loop **then**
9: $\quad \langle \cdot, \cdot, \overline{\Pi} \rangle = \mathsf{invariant}(\upsilon, \ell' \to \ldots \to \ell)$
10: $\quad \langle \overline{\Psi}, \sigma^{\omega} \rangle = \langle \overline{\Pi}, \sigma^{\omega} \rangle$ and **goto** 12
11: $\langle \overline{\Psi}, \sigma^{\omega}, \mathsf{change} \rangle = \mathsf{UnwindTree}_V(\upsilon, \sigma^{\omega})$
12: let υ be annotated with $\langle \cdot, \sigma^{\omega}{}_{old} \rangle$
13: label υ with $\langle \overline{\Psi}, \sigma^{\omega} \rangle$ and **return** $\langle \overline{\Psi}, \sigma^{\omega}, \mathsf{change} \vee \neg(\sigma^{\omega}{}_{old} \sqsubseteq_{\mathcal{D}^{\omega}} \sigma^{\omega}) \rangle$

$\mathsf{UnwindTree}_V(\upsilon \equiv \langle \ell, s, \Pi \rangle, \sigma^{\omega}{}_{in})$
1: $\overline{\Psi} = \mathit{true}$, $\sigma^{\omega} = \sigma^{\omega}{}_{in}$, $\mathsf{change} = \mathit{false}$
2: **foreach** transition relation $\ell \xrightarrow{\mathsf{op}} \ell'$
3: $$\upsilon' \triangleq \begin{cases} \langle \ell', s, \Pi \wedge [\![c]\!]_s \rangle & \text{if op} \equiv \mathsf{assume(c)} \\ \langle \ell', s[x \mapsto S_x], \Pi \wedge [\![x = e]\!]_s \rangle & \text{if op} \equiv \mathsf{x = e} \text{ and } S_x \text{ fresh variable} \end{cases}$$
4: $\quad \langle \overline{\Psi}', \sigma^{\omega\prime}, \mathsf{c} \rangle = \mathsf{BackwardDeps}_V(\upsilon', \sigma^{\omega}{}_{in})$
5: $\quad \overline{\Psi} = \overline{\Psi} \wedge \widehat{wlp}(\mathsf{op}, \overline{\Psi}')$
6: $\quad \sigma^{\omega} = \sigma^{\omega} \sqcup_{\mathcal{D}^{\omega}} \widehat{pre}_{\mathcal{D}^{\omega}}(\sigma^{\omega\prime}, \mathsf{op}, s)$
7: $\quad \mathsf{change} = \mathsf{change} \vee \mathsf{c}$
8: **return** $\langle \overline{\Psi}, \sigma^{\omega}, \mathsf{change} \rangle$

$\mathsf{BackwardDepsLoop}_V(\upsilon, \sigma^{\omega})$
1: $\sigma^{\omega\prime} = \sigma^{\omega}$, $\mathsf{change} = \mathit{false}$
2: **do** $\langle \cdot, \sigma^{\omega\prime}, \mathsf{change} \rangle = \mathsf{BackwardDeps}_V(\upsilon, \sigma^{\omega\prime})$ **while** (change)

Fig. 3. Path-Sensitive Backward Slicing Analysis

Line 1 initializes the (local) variable change to *false*, which will be updated later. Next, the three base cases for symbolic states are handled - infeasible, terminal, and reuse:

- In line 2, the function $\mathsf{INFEASIBLE}(\langle \cdot, \cdot, \Pi \rangle)$ checks whether Π is satisfiable. If not, the symbolic execution detects an infeasible path and halts, excluding any dependency which would have been inferred from the non-executable path. In addition, it produces an interpolant from Π and *false*, namely $\overline{\Psi} \equiv \mathit{false}$, which generalizes the current path condition ($\Pi \models \overline{\Psi}$ and $\overline{\Psi}$ is *false*). Since the path is not executable there is no variable that may affect the slicing criterion and hence, the set of dependencies returned is empty.
- In line 3, the function $\mathsf{TERMINAL}(\langle \ell, \cdot, \cdot \rangle)$ checks if the symbolic state is a terminal node by checking if $\ell = \ell_{\mathsf{end}}$. If yes, the execution has reached the end of a path. Since the path is feasible, it can be fully generalized returning the interpolant $\overline{\Psi} \equiv$

true. Since ℓ is a terminal node, the set of dependencies is the set of variables in the slicing criterion, V. The witness formula for each variable from V is initially *true*.
- In line 4 the algorithm searches for another state υ' whose dependencies can be reused by the current state υ so that the symbolic execution can be stopped. For this, the function $\mathsf{REUSE}(\upsilon, \upsilon')$ tests the reusing conditions in Eq. 6. If the test holds, the state υ can reuse the dependencies computed by υ'. Note that the amount of search space pruned by our method depends on how often this case is triggered.

If all three base cases are not applicable, the algorithm unwinds the execution tree by calling the procedure $\mathsf{UnwindTree}_V$ at line 11. $\mathsf{UnwindTree}_V$, at line 3, executes one symbolic step [6] and calls the main procedure $\mathsf{BackwardDeps}_V$ with the successor state (line 4). After the call, the two key remaining steps are to compute:

- the interpolant $\overline{\Psi}$ ($\mathsf{UnwindTree}_V$ line 5) that generalizes the symbolic execution tree below υ while preserving its infeasible paths. The procedure $\widehat{wlp} : Ops \times FOL \to FOL$ ideally computes the *weakest liberal precondition (wlp)* [7] which is the weakest formula on the initial state ensuring the execution of op results in a final state $\overline{\Psi}'$. In practice, we approximate *wlp* following the algorithm described in [14].[7] The interpolant $\overline{\Psi}$ is an FOL formula consisting of the conjunction of the result of $\widehat{wlp}$ on each child's interpolant.
- the solution, σ^{ω}, for the current state υ at line 6 which is computed by executing $\widehat{pre}_{\mathcal{D}^{\omega}}$ on each child's solution and then combining all solutions using $\sqcup_{\mathcal{D}^{\omega}}$.

In addition, at line 7 it also records changes in any child's symbolic state (if any) and then returns a triple in the same format as $\mathsf{BackwardDeps}_V$'s return value. In $\mathsf{BackwardDeps}_V$, line 12 updates change to *true* if either it was set to *true* in $\mathsf{UnwindTree}_V$ at line 11 or the current symbolic state is about to be updated with a more precise solution than that it already has. The final operation before returning from $\mathsf{BackwardDeps}_V$ is to label the state υ with the reusing conditions and solution (line 13).

Now we continue describing our algorithm by discussing how it handles loops. The main issue is to produce a finite symbolic execution tree on which a fixpoint of the dependencies can be computed.

For this, the algorithm in Fig. 3 takes an annotated transition system in which program points are labelled with inductive invariants inferred automatically by an abstract interpreter using an abstract domain such as *octagons* or *polyhedra* (we borrow the ideas presented in [17] for this purpose). We assume the abstract interpreter provides a function $\mathsf{getAssrt}$ which, given a program location ℓ and a symbolic store s, returns an assertion in the form of an FOL formula renamed using s, which holds at ℓ. Note that when applied at loop headers, $\mathsf{getAssrt}$ will return a loop invariant. However, we would

---

[6] Note that the rule described in line 3 is slightly different from the one described in Sec. 3 because no consistency check is performed. Instead, the consistency check is postponed and done by the first base case at line 2.

[7] Current SMT solvers (e.g. [4]) can produce (very efficiently) interpolants at each location along a path from a single query which can be used for approximating wlp's. However, those interpolants are often stronger than those generated by [14].

like to strengthen it using the constraints propagated from the symbolic execution. The function invariant performs this task as follows:

$$\mathsf{invariant}(\langle \ell, s, \Pi \rangle, \ell_1 \to \ell_n) \triangleq \begin{cases} \textbf{let } s' = \mathsf{havoc}(s, \mathsf{modifies}(\ell_1 \to \ell_n)) \\ \quad \overline{\Pi} = \mathsf{getAssrt}(\ell, s') \wedge \Pi \\ \textbf{in } \langle \ell, s', \overline{\Pi} \rangle \end{cases}$$

$\mathsf{havoc}(s, Vars) \triangleq \forall v \in Vars \bullet s[v \mapsto z]$
where z is a fresh variable (implicitly $\exists$-quantified).

$\mathsf{modifies}(\ell_1 \to \ldots \to \ell_n)$ takes a sequence of transitions and returns the set of variables that may be modified during its symbolic execution.

Intuitively, invariant clears the symbolic store of all variables modified in the loop (using the havoc function) and then enhances the path condition Π of the symbolic state with the invariants from the abstract interpreter.

Let us now explain the *grey* boxes in Fig. 3. Lines 5-7 in $\mathsf{BackwardDeps}_V$ cover the case when a loop header has been encountered. The objective is to abstract the current symbolic state by using the loop invariant obtained from the abstract interpreter. The algorithm calls the function invariant (at line 6) with the transitions in the loop so as to obtain a copy of the current symbolic state annotated with the approximate loop invariant in its path condition. At line 7, the $\mathsf{UnwindTree}_V$ procedure is called on the resulting abstracted symbolic state to explore the symbolic subtree associated with the loop.

If the symbolic execution encounters a loop backedge (lines 8-10) from ℓ to ℓ' it halts and backtracks. The reason is that the loop header at ℓ' has already been symbolically executed with a loop invariant. Hence there is no need to continue the loop since the invariant ensures that no new feasible paths will be encountered if it is explored again. This is our basic mechanism to make the symbolic execution of the loop finite.

Finally, the main algorithm to handle loops, $\mathsf{BackwardDepsLoop}_V$, makes calls to the function $\mathsf{BackwardDeps}_V$ until there is no change detected in the symbolic state of any program point. We present it in its simplest form, but it can be easily optimized to call $\mathsf{BackwardDeps}_V$ only with those loop transitions affected by a change.

5 Results

We implemented a proof-of-concept prototype as an extension to TRACER [14]. TRACER is a software verifier for C programs from which we used mainly its symbolic execution interpreter and its capabilities for computing interpolants from infeasible paths.

Our prototype augmented TRACER in different ways. Given an operation that involves pointers our prototype updated the sets *def* and *use* to accommodate the points-to information correctly. For instance, given the statement `*p =*q` the set *def* contains everything that might be pointed to by `p` and the set *use* includes everything that might be pointed by `q`. Regarding loops, programs were first annotated with loop invariants[8]

[8] We tried several numerical abstract domains with different tradeoffs between performance and precision (e.g., octagons and polyhedra) but obtained same results. As a limitation, those invariants cannot express properties about heap-allocated data structures.

Program	LOC	Path-Insens		Path-Sens	
		Size Red	Time	Size Red	Time
mpeg	5K	4%	21s	8%	628s
diskperf	6K	32%	2s	57%	94s
floppy	8K	36%	9s	47%	263s
cdaudio	9K	23%	10s	52%	301s
serial	12K	39%	16s	50%	395s
fcron.2.9.5	12K	42%	32s	61%	832s
Mean		**23%**	**15s**	**38%**	**418s**

Table 1. Results on Intel 3.2Gz 2Gb evaluating path-sensitiveness

provided by the abstract intepreter InterProc [15] ensuring that symbolic execution is finite. Then, we implemented a fixpoint algorithm operating over symbolic execution trees that computes dependencies among variables following [19]. Witness paths were represented as formulas (conjunction of the constraints along the path) and stored efficiently in order to increase sharing among them. Functions were inlined and external functions were modeled as having no side effects and returning an unknown value.[9]

We used several instrumented device driver programs previously used as software model checking benchmarks: cdaudio, diskperf, floppy, and serial. In addition, we also considered mpeg, the mpeg-1 algorithm for compressing video, and fcron.2.9.5, a cron daemon. For the slicing criterion we consider variables that may be of interest during debugging tasks. For the instrumented software model checking programs, we choose as the slicing criterion the set of variables that appear in the safety conditions used for their verification in [10]. In the case of mpeg we choose a variable that contains the type of the video to be compressed. Finally, in fcron.2.9.5 we choose all the file descriptors opened and closed by the application.

Table 1 compares our path-sensitive slicer (columns labelled with Path-Sens) against the same slicer but without path-sensitivity (labelled with Path-Insens). Path-insensitivity is achieved by the following modifications in our slicer: (1) considering all paths as feasible, and (2) always forcing reuse. These changes have the same effect as always merging the abstract states along incoming edges in a control-flow merging node. In other words, they mimic running a path-insensitive slicer on the original CFG. We could have used a faster off-the-shelf path-insensitive program slicer (using e.g., [11]), however, our objective here is to isolate the impact of path-sensitivity and hence, we decided to perform the comparison on a common platform to produce the fairest results.

The column LOC represents the number of lines of program without comments. The column Size Red shows the reduction in slice size (in %) wrt the original program size. The reduction size is computed using the formula $(1 - \frac{size\ of\ slice}{size\ of\ original}) \times 100$. By *size* we mean all executable statements in the program, excluding type declarations, unused functions, comments, and blank lines. A minor complication here is that the SET may contain multiple *instances* of program points in the CFG, as can be seen in Fig. 1(c). To compare the reduction in slice sizes fairly, we use the rule mentioned at the beginning of Sec. 4 to compute slices: a transition in the original CFG is included in the slice if any

[9] It is well-known that function inlining can be very inefficient and in fact, not possible in the presence of recursive functions. However, performing an interprocedural path-sensitive analysis is beyond the scope of this paper.

of its instances in the SET is included in the slice. The column Time reflects the running time of the analysis in seconds excluding the external abstract interpreter. Finally, we summarize in row Mean the numbers of columns Size Red and Time by computing their *geometric* and *arithmetic* mean, respectively.

Our experimental evaluation shows that our path-sensitive slicer improves significantly in terms of size reduction over its path-insensitive counterpart. Roughly, slices produced by Path-Sens are 38% smaller than the original programs while only 23% in the case of Path-Insens. The mpeg program is an exception since the size of the slices in both Path-Insens and Path-Sens are quite big (i.e., very small reduction). The reason is that in mpeg all the computations depend on the type of video to be compressed which is our slicing criterion. On the other hand, the running times of Path-Sens (with a mean of 418 secs) are reasonable considering the size of the programs and the current status of our prototype implementation which has significant room for improvement. The analysis of mpeg is especially slow and it is due to the existence of many nested loops which are not supported efficiently by our naive fixpoint implementation.

To emphasize the importance of our reuse technique based on interpolation and witnesses we experimented with two variants of our algorithm. We first ran our path-sensitive slicer without reuse which mimics *Conditioned Slicing* [3] (see Sec. 6 for more details). Our second variant replaced interpolants with a syntactic method avoiding using the solver. Given formulas A and B (the inputs to the interpolation algorithm) the reusing condition is a formula formed from A such that any constraint *syntactically independent* from B is removed (taking into account the transitive closure of constraint dependencies). Then, the REUSE procedure can be implemented as a subset operation rather than an entailment test. Interestingly, neither of these two variants was able to finish with any program after a timeout of 1 hour or memory consumption of 2.5 Gb.

6 Related Work

Static slicing remains a very active area of research. We limit our discussion to the most relevant works that take into account path-sensitiveness. We also discuss pruning techniques that might have influenced our work.

Fully path-sensitive methods. Conditioned slicing [3] also performs symbolic execution excluding infeasible paths before applying a static slicing algorithm, and hence it is fully path-sensitive (for loop-free programs) similar to us. However, even efficient implementations (e.g., [6]) still perform full path enumeration and essentially explore the search space of the naive SET suffering from the path explosion problem.

Partially path-sensitive methods. A more scalable but not fully path-sensitive approach is described by Snelting et al. [18]. They compute the dependency between two program points y and x using the Program Dependence Graph (PDG) [11] and apply the following rule to remove spurious dependencies: $I(y,x) \Rightarrow \exists \overline{v} : PC(y,x)$, where $I(y,x)$ stands for y *influences* x (i.e., there is a dependency at x on y), $\overline{v}$ is some assignment of values to program variables and $PC(y,x)$ is the path condition from y to x. Essentially it means that if the path condition from y to x is found to be unsatisfiable, then there is definitely no influence from y to x. If there are multiple paths between two points, the path condition is computed as a disjunction of each path.

For the program in Fig. 1(a), Snelting et al. would proceed as follows. In the PDG there will be a dependency edge from ℓ_8 to ℓ_1, hence they would check to see if the path condition $PC(1,8)$ is unsatisfiable. First they calculate the path condition from ℓ_4 to ℓ_8 as $PC(4,8) \equiv (x = 1 \wedge y > 0 \wedge z = x) \vee (y = 0 \wedge y > 0 \wedge z = x) \equiv (x = 1 \wedge y > 0 \wedge z = x)$. Now they use this to calculate $PC(1,8) \equiv (x = 0 \wedge y = 5 \wedge ((a > 0 \wedge b = x + y \wedge PC(4,8)) \vee (a \leq 0 \wedge PC(4,8))))$[10], which is not unsatisfiable. Hence the statement x=0 at ℓ_1 will be included in the slice. The fundamental reason for this is that for them, path conditions are only necessary and not sufficient, so false alarms in examples such as the above are possible. An important consequence of this is the fact that even for loop-free programs, their algorithm cannot be considered "exact" in the sense described at the end in Sec. 1. However, our algorithm guarantees to produce no false alarms for such programs.

Another slicer that takes into account path-sensitiveness up to some degree is Constrained slicing [8] which uses *graph rewriting* as the underlying technique. As the graph is rewritten, modified terms are tracked. As a result, terms in the final graph can be tracked back to terms in the original graph identifying the slice of the original graph that produced the particular term in the final graph. The rules described in [8] mainly perform constant propagation and dead code detection but not systematic detection of infeasible paths. More importantly, [8] does not define rules to prune the search space.

Interpolation and SAT. Interpolation has been used in software verification (e.g., [1, 10, 16, 12, 14]) as a technique to eliminate facts which are irrelevant to the proof. Similarly, SAT can explain and record failures in order to perform conflict analysis. By traversing a reverse implication graph it can build a *nogood* or conflict clause which will avoid making the same wrong decision. Our algorithm has in common the use of interpolation that can be seen also as a form of *nogood learning* in order to prune the search space. But this is where the similarity ends. A fundamental distinction is that in program verification there is no solution (e.g., backward dependencies) to compute/discover and hence, there is no notion of reuse and the concept of witness paths does not exist. The work of [9] uses interpolation-based model checking techniques to improve the precision of dataflow analysis but still for the purpose of proving a safety property.

Finally, a recent work of the authors [13] has been a clear inspiration for this paper. [13] uses interpolation and witnesses as well to solve not an analysis problem, but rather, a *combinatorial optimization* problem: the Resource-Constrained Shortest Path (RCSP) problem. Moreover, there are other significant differences. First, [13] is totally defined in a finite setting. Second, [13] considers only the narrower problem of extraction of bounds of variables for loop-free programs while we present here a general-purpose program analysis like slicing. Third, this paper presents an implementation and demonstrates its practicality on real programs.

7 Conclusions

We presented a fully path-sensitive backward slicer limited only by solving capabilities and loop invariant technology. The main result is a symbolic execution algorithm

[10] We have simplified this formula since Snelting et al. use the SSA form of the program and add constraints for Φ-functions, but the essential idea is the same.

which avoids imprecision due to infeasible paths and joins at merging points and halts execution of a path if certain conditions hold while reusing dependencies from already explored paths. The conditions are based on a notion of interpolation and witness paths with an aim to detect "a priori" whether the exploration of a path could improve the accuracy of the dependencies computed so far by other paths. We demonstrated the practicality of the approach with real medium-size C programs.

Finally, although this paper targets slicing, our approach can be generalized and applied to other backward program analyses (e.g., Live Variable, Very Busy Expressions, Worst-Case Execution Time analysis, etc.) providing them path-sensitiveness.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI'01*, pages 203–213.
2. L. Bent, D. C. Atkinson, and W. G. Griswold. A Comparative Study of Two Whole Program Slicers for C. Technical report, University of California at San Diego, La Jolla, 2001.
3. G. Canfora, A. Cimitile, and A. De Lucia. Conditioned Program Slicing. *Information and Software Technology*, 40, no. 11-12:595–607, 1998.
4. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolant Generation in Satisfiability Modulo Theories. In *TACAS'08*, pages 397–412.
5. W. Craig. Three Uses of Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *Journal of Symbolic Computation*, 22, 1955.
6. M. Daoudi, L. Ouarbya, J. Howroyd, S. Danicic, M. Harman, C. Fox, and M.P. Ward. Consus: A Scalable Approach to Conditioned Slicing. In *WCRE'02*, pages 109-118.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.
8. J. Field, G. Ramalingam, and F. Tip. Parametric Program Slicing. In *POPL '95*, pages 379–392.
9. J. Fischer, R. Jhala, and R. Majumdar. Joining Dataflow with Predicates. In *ESEC/FSE-13*, pages 227–236, 2005.
10. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from Proofs. In *POPL'04*, pages 232–244.
11. S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Dependence Graphs. In *PLDI '88*, pages 35–46.
12. J. Jaffar, , J.A. Navas, and A. Santosa. Unbounded Symbolic Execution for Program Verification. In *RV'11*, pages 396–411.
13. J. Jaffar, A. E. Santosa, and R. Voicu. Efficient Memoization for Dynamic Programming with Ad-hoc Constraints. In *AAAI'08*, pages 297–303.
14. J. Jaffar, Vijayaraghavan Murali, J.A. Navas, and A. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV'12*, pages 758–766.
15. G. Lalire, M. Argoud, and B. Jeannet. The Interproc Analyzer. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc, 2009.
16. K. L. McMillan. Lazy Annotation for Program Testing and Verification. In *CAV'10*, pages 104-118.
17. S Seo, H Yang, and K Yi. Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In *APLAS'03*, pages 230–245.
18. G Snelting, T Robschink, and J Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. In *TOSEM'06*, volume 15, pages 410–457.
19. M. Weiser. Program Slicing. In *ICSE'81*, pages 439–449.