A Path-Sensitively Sliced Control Flow Graph

Joxan Jaffar
National University of Singapore, Singapore
joxan@comp.nus.edu.sg

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

ABSTRACT

We present a new graph representation of programs with specified target variables. These programs are intended to be processed by third-party applications querying target variables such as testers and verifiers. The representation embodies two concepts. First, it is path-sensitive in the sense that multiple nodes representing one program point may exist so that infeasible paths can be excluded. Second, and more importantly, it is sliced with respect to the target variables. This key step is founded on a novel idea introduced in this paper, called "Tree Slicing", and on the fact that slicing is more effective when there is path sensitivity. Compared to the traditional Control Flow Graph (CFG), the new graph may be bigger (due to path-sensitivity) or smaller (due to slicing). We show that it is not much bigger in practice, if at all. The main result however concerns its quality: third-party testers and verifiers perform substantially better on the new graph compared to the original CFG.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*

General Terms

Algorithms, Reliability, Verification

Keywords

Symbolic execution, program slicing, program transformation

1. INTRODUCTION

We present a new intermediate graph representation for C programs with specified target variables. These programs are intended to be processed by third-party applications such as verifiers and testers. The representation embodies two concepts. First, it is *path-sensitive* in the sense that there may be multiple nodes representing one program point so that infeasible symbolic execution paths can be excluded. Second, and more importantly, the graph is *sliced* with respect to the target variables.

FSE'14 , November 16–22, 2014, Hong Kong, China

We begin with a promotional example: consider the C program

```
if(c) p = 1; else p = 0;
x = 0;
if(p > 0) x = 1;
if(x == 0) z = 1;
TARGET: {z}
```

Note that no *static slicing* is effective on this program because each statement and variable affects the target `z` along at least one path. However, we can *transform* this program into an equivalent one:

```
if (!c) z = 1;
```

which produces, on any given input, the same values for `z` as the original program. Clearly this transformed program would be more efficient when input to a verifier or tester which seeks properties of `z`. We arrived at this transformation as follows. Let S denote the program fragment comprising of all but the first if-statement of the original program. Now consider slicing S in the *context* $p = 1$ (the "then" body of the first if-statement). Clearly S would not modify the variable `z` because only the statements `x=0` and `x=1` would be executed in this context. Next consider the alternative context $p = 0$ (the "else" body). Now S would execute the statements `x=0` and `z=1`, from which the former can be sliced away. Hence we get the transformed program.

In other words, we arrived at this new program by first considering path-sensitivity, and more specifically, the original program's *symbolic execution (SE) tree*. The general idea is that slicing of a program fragment can be much more effective when it is done with a given context. The SE tree in fact displays the context of a program fragment as it unfolds through the various paths that bring execution to this fragment. Now consider the example:

```
if(c) p = 1; else p = 0;
S
TARGET: {z}
```

where S now represents a program fragment which *cannot* be sliced by restricting consideration of the values of `z` at the end of the program. That is, all symbolic execution paths in S produce some (different) output value in `z` at the end, regardless of the initial values of `c` or `p`. Here, by being path-sensitive, we would produce a CFG that corresponds to the program:

```
if(c) { p = 1; S; }
else { p = 0; S; }
```

This new program is effectively twice the size of the original program due to the duplication of `S`, and yet there is no benefit from using this enlarged representation.

It is folklore that a fully path-sensitive representation of symbolic execution is simply intractable, for the representation doubles

in size for each branch statement encountered. The only alternative is to have some form of *merging* where, at some stages in the construction of the graph, certain paths in the graph transition into the *same* node. If the merging is performed at every opportunity, the original CFG would be obtained. If not performed at all, the full, possibly intractable, SE tree would be obtained. The big question is, therefore, *how much* merging is needed?

In this paper, we present a method for producing a path-sensitive CFG by constructing a SE tree but merging nodes *exactly when* the merge does not hide any information that affects slicing. That is, when our algorithm merges a node in the tree with another, it guarantees that had the node been symbolically executed instead, one would obtain precisely the same slicing information as that of the node it's being merged with. A key step involved in the construction of our CFG, which we call the *Path-Sensitively Sliced CFG* or *PSS-CFG*, is "Tree Slicing", a powerful technique to merge and slice arbitrarily different SE sub-trees under certain conditions.

Our main result is that the PSS-CFG, when "decompiled" (or transformed) into regular programs that can be directly used by applications which query target variables (e.g., a concolic tester that targets the program's outputs, or a verifier with a safety property), produces significant improvement in terms of time usage as compared to using the original program. The strength of the PSS-CFG is that it can be used "out-of-the-box" by a wide number of third-party software engineering applications. We consider two main applications - program testing and verification, and show in Section 6 how they can benefit from the PSS-CFG to gain in performance.

2. RELATED WORK

With regards to performing an "offline" program transformation for general use, our closest related work is [3], performing a path-sensitive transformation with an aim to "improve the path-insensitive analysis used inside the F-SOFT tool to obtain the effects of path-sensitive analysis". The main difference is that their transformation only removes infeasible paths from the CFG without performing slicing. As a result, it is not clear how the performance of a verifier or tester could be improved because they are themselves path-sensitive and would not consider the infeasible paths anyway. Hence, their target application is a *path-insensitive* analyser within F-SOFT that can benefit from the removal of infeasible paths, as it would spuriously consider them otherwise. Nevertheless we share with them the high level goal of performing an "offline" program transformation that helps an external application.

Since program transformation is a specific area, we also discuss related work that do not perform transformation but still provide benefits of path-sensitivity for external consumption. In this regard, another related work is [6] that discards irrelevant tests in concolic testing by tracking values read and written by executed program statements. Then, when two states differing only in program values *not subsequently read* are visited, the exploration of the second state can be pruned. The main difference is that they use live range information of variables to make the decision about pruning paths, which results in lesser pruning compared to our method that uses slicing (specifically, dependency information). Moreover they do not perform program transformation themselves, but work with the concolic tester to discard certain tests during execution. Thus, they are dependent on the external application, whereas we perform an offline transformation that is independent of the application.

Another related work is [17] that performs slicing of paths, but their goal is to reduce the size of the counterexample path generated by CEGAR-based verifiers during their process. As a result, they do not work with the entire program's CFG and hence, they are not concerned with splits and merges in the CFG. However a program transformation algorithm like ours needs to work with the whole CFG to decide where to split or merge. Slicing as a precursor to model checking has also been shown to be effective in [12].

The recent work [18] performs dynamic state merging during symbolic execution in order to combine paths. While reducing the number of paths, the formulas corresponding to merged states are more complicated. They showed that their chosen heuristic for deciding which states are merged produced significant speedups. A similarity to our paper here is that we also perform state merging but we do so by learning when different states need not be explored. Also, [18] does not consider slicing, but our algorithm merges only when it can guarantee lossless-ness of slicing (dependency) information. State merging in the context of *dynamic* symbolic execution has also been studied in a more recent work [2].

We will see in Section 5 that our method uses *interpolation* to perform state merging. Interpolation has been used in program testing and verification [19, 16, 20, 13] to reduce state space and contain "path-explosion". However the similarity with our paper is only in the use of interpolation for state merging. Our method has to additionally guarantee that the merging is *lossless*, something that is inapplicable to these works. Also, in Section 6, we experimentally evaluate the PSS-CFG using applications of testing and verification, which may be another source of confusion with these works. The fundamental difference is that these works involve directly performing the testing or verification process on the program. We simply *evaluate* the PSS-CFG on third-party testing and verification tools to show that they benefit in performance (in fact, [19] is one of the verifiers we use). But the PSS-CFG is a much more generic object not limited to just testing and verification.

We finally mention the work [15] which performed static slicing. The technical approach there was to first generate a path-sensitive symbolic execution tree which then was used to determine which primitive program statements could be sliced. It performed slicing on the program, using the symbolic tree to slice a statement that does not affect the target *anywhere* in the tree. In contrast, here we perform slicing *on the tree itself* to transform it into a new tree using the transformation rules. Our method allows the same program statement to be sliced form one part of the tree but not another, a scenario in which [15] simply cannot slice the statement from the program at all. This fundamental difference will be exposed in the example in Section 3. Moreover, a key technical slicing step of our method, called "Tree slicing", involves slicing a compound statement *from a tree*, a problem not relevant to [15]: in general, the symbolic execution subtrees rooted at what corresponds to the end of a compound statement may not be identical. A main theoretical result of this paper, concerning the correctness of the transformation (Theorem 2), is that under certain conditions formalised in Theorem 1, we can correctly slice away the entire compound statement, and *merge* the following subtrees even though they are different, while still retaining the necessary equivalent behavior of the original program on the target variables.

3. BASIC IDEA

Consider the program in Fig. 1, where a `read` call signals a concolic tester to generate an input, and the target of interest is the variable `z` in the end. The program has two inputs `c` and `d` which decide the control flow and 8 paths to traverse. At the outset, note that no static slicers, even path-sensitive ones like [15] or the well-known Frama-C [8], are effective on this program because each statement along *some* path affects the target.

Our algorithm has two steps: first, it performs symbolic execution to generate the SE tree annotated with dependency information at each node (Fig. 1(b)). The goal here is to be as path-sensitive

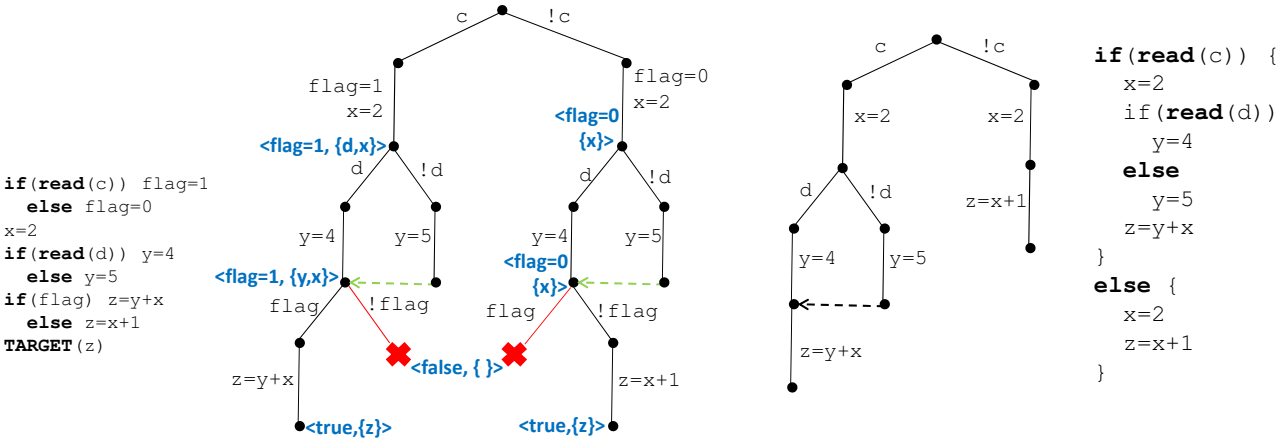

Figure 1: (a) A program (b) its symbolic execution tree (c) its PSSCFG and (d) the corresponding decompiled program

as possible since it makes dependency information more precise. However since path-sensitivity also makes the SE tree exponential in the number of branches, we try to keep its size in check by *merging* while ensuring this does not cause imprecision of dependencies. In the second step, transformation rules are applied on the SE tree to get the final PSS-CFG (Fig. 1(c)). These rules take advantage of the precise dependency information obtained in the first step.

In Fig. 1(a), our algorithm first encounter a branch on `c`. To be path-sensitive, it splits symbolic execution into two – one with context `c` and the other with context `!c`. In a DFS fashion, it first explores the context `c`, symbolically executing the statements `flag=1` and `x=2` and as usual, carrying these path constraints in a logic formula. Upon reaching the next branch, it again splits into two – with context `d` and `!d`. Continuing along the context `d` it executes `y=4` and reaches the final branch. Again it splits into two – with context `flag` and `!flag`, following the former and finally executing `z=y+x` before reaching the terminal point.

At this point the path formula is represented by the constraints: $c \wedge flag = 1 \wedge x = 2 \wedge d \wedge y = 4 \wedge flag \wedge z = y + x$[1] which is satisfiable, meaning the path is feasible. Our algorithm now generates the backwards dependency information for this feasible path, resulting in the dependency set $\{z\}$, the target at the terminal point. This is then propagated back to the branch point on `flag` by applying Weiser's formulas [23], resulting in the set $\{y, x\}$. In addition to the dependency set, our algorithm also computes the *witness path* for each variable in the set. A witness path for a variable at a particular node in the SE tree is a path from that node along which the variable affects the target in the end, i.e., it is a "witness" for the variable affecting the target. The witness path for the set $\{y, x\}$ is the path executing `flag` and `z=y+x`, corresponding to the formula $flag \wedge z = y + x$. To avoid cluttering, we do not show the witness paths in Fig 1(b). Witness paths are needed to ensure there is no loss of dependency information when we merge two nodes later.

Next, our algorithm backtracks and explores the other context from the last split point: `!flag`. The path formula $c \wedge flag = 1 \wedge x = 2 \wedge d \wedge y = 4 \wedge \neg flag$ is unsatisfiable, hence the path is infeasible. Now it computes an *interpolant*, a formula that captures the essence of infeasibility of the path at the branch point. The purpose of the interpolant is to exclude irrelevant information pertaining to the infeasibility so that merging of another node with the current one is more likely to happen in future. For the above path, one possible interpolant at the branch point is $flag = 1$ which is enough to capture the inconsistency with $\neg flag$. There are numerous methods to compute interpolants (e.g., weakest preconditions) and the quality of the interpolant will affect the amount of merging performed by our algorithm. The interpolant at a terminal node is $true$ and the interpolant at an infeasible node is $false$, as shown.

In summary, our method computes at each node in the SE tree: (a) the dependency sets (b) witness paths from feasible paths (c) interpolants from infeasible paths. At the branch point on `flag`, these are $\{y, x\}$, $flag \wedge z = y + x$ and $flag = 1$, respectively. The astute reader might have noticed that our algorithm did not include the variable `flag` in the dependency set, whereas a traditional slicer such as [8, 15] would have included it due to control dependency. The reason is that along this particular path only one branch, the one where `flag` is true (non-zero), is feasible and the other is infeasible. That is, `flag` is always true at this point along this path. Hence the value of `flag` does not affect the execution of the statement `z=y+x`, therefore it is not control-dependent on `flag`. However, `flag` being true is needed to *preserve* the infeasibility of that branch and this is captured in the interpolant.

Next our algorithm backtracks again to the previous split point to explore the other branch: `!d`. It executes the statement `y=5` and reaches the branch on `flag`, this time under the different context `!d`. Now, the most important step of checking whether the current context of the branch can be merged with the previously explored context is performed. First, our algorithm makes sure the merging is *correct* by checking if current path formula $c \wedge flag = 1 \wedge x = 2 \wedge \neg d \wedge y = 5$ implies the interpolant $flag = 1$. This check succeeds, meaning that this context can be merged soundly with the previous one[2]. Next it makes sure the merging will incur no loss of *precision* by checking if the witness path from the previous context $flag \wedge z = y + x$ is feasible in the current context. Indeed it is, as the current path formula is consistent with the witness formula. Therefore, the current context of the branch on `flag` is merged with the previous context without any loss of precision. This is formalised in Theorem 1, that says if both checks succeed, then by exploring the current context of the node one would obtain *exactly* the same dependency information as the node it's being merged with. The merging is denoted by the green dashed arrow in Fig 1(b).

[1] We omitted the `read` calls, which only the tester understands.

[2] The concept of soundness is formalised later in Lemma 1.

Our algorithm now propagates backwards the dependency sets, interpolants and witness paths to the previous branch on `d`, resulting in the set $\{d, x\}$, interpolant $flag = 1$, and witness path $d \wedge y = 4 \wedge flag \wedge z = y + x$. Note that this time, it considered the control-dependence of `y` on `d` as both paths from the `d` branch were feasible, thus adding `d` to the dependency set. It then backtracks to the first split point on `c` and explores the other branch `!c`. Upon reaching the branch on `d` again, it tries to merge with the previous context of the branch by checking if the current path formula $\neg c \wedge flag = 0 \wedge x = 2$ implies the interpolant $flag = 1$. It does not, so the merging cannot be performed and so it proceeds to explore the rest of the tree under the node, resulting in the final SE tree as shown in Fig. 1(b). We do not explain the process again as it is very similar to the left half of the SE tree. However there are a few important things to note in the final tree. The branch on `d` is duplicated due to the split at the previous branch on `c`. However under the context `!c`, the dependency set at the branch point on `flag` is only $\{x\}$ as opposed to $\{y, x\}$ under the context `c`, because here there is no data-dependency of `z` on `y`. This is the advantage of path-sensitivity – we have obtained a more precise dependency information at a different context of the same program point by considering the contexts separately, although at the price of duplication of the `d` branch. However we will see soon that because of the more precise information, the duplication can be controlled by slicing.

In phase two of our algorithm, we apply three transformation rules that will process the SE tree annotated with dependency information to transform it into the final PSS-CFG. We give an informal description of each rule here, as they are formalised in Section 5.2.

- **Rule 1**, the traditional slicing rule, states if the LHS of an assignment statement does not occur in the dependency set after it, the statement can be removed.
- **Rule 2** states that if a branch point has only one feasible path arising from it, the branch point can be removed. The reasoning is that if a branch point has only one feasible path from it, then in that particular context the branch condition can be deterministically evaluated to true (or false). Thus it can simply be replaced with the "then" (or "else") body.
- **Rule 3** (called "Tree Slicing"), which is more powerful in reducing the PSS-CFG's size, states that an entire branch is irrelevant to the target and can be removed if both the "then" and "else" bodies contain no statement that is included in the slice. This rule is more complicated than it seems at first because working with trees, a problem arises when we remove a branch point: conceptually there could be *two* subtrees whose parent, the branch point, is about to be removed. The two sub-trees could be arbitrarily different because of the different contexts leading into them. Which one should be linked to the branch point's parent? The rule guarantees that regardless of which sub-tree is picked, the transformation is still sound, *provided* that our algorithm declared the sub-trees to be merged. This important non-trivial result is formalised in Section 5.2, and is one of the many fundamental differences between our transformation method and *static slicing* methods, that slice on the program, not the tree.

Note that in general, the rules are not limited to the above three, and one can indeed formulate more sophisticated rules. For instance, *amorphous slicing* [11] can be applied to further elide statements from the SE tree, where it is more likely to be useful compared to applying on the original CFG, as the SE tree exposes different symbolic paths leading to a program point. For our benchmarks, however, the above three rules were sufficient to provide benefit. These rules are applied on the SE tree until none of them can be applied any more, and the resultant graph is the PSS-CFG.

In our example, applying Rule 1 on the SE tree removes the statements `flag=1` and `flag=0`. Applying Rule 2 removes the two branches on `flag` that have an infeasible path. More interesting is the application of Rule 3. It cannot be applied on the `d` branch under the context `c` because in that context, `y=4` and `y=5` are included in the slice (the dependency set after the branch is $\{y, x\}$). However, it can indeed be applied on the `d` branch under the context `!c` because neither `y=4` nor `y=5` is included in the slice (the dependency set after the branch in this context is only $\{x\}$), and our algorithm had merged the symbolic state after its "then" and "else" body. Thus, Rule 3 removes the `d` branch under the context `!c` but not in the context `c`, to get the final PSS-CFG in Fig. 1(c). This reduction of the graph due to slicing complements the blow-up due to path-sensitivity, and is critical to maintaining the PSS-CFG's size. Finally, note that Rule 3 cannot be applied on the top-level `c` branch because the two subtrees after its "then" and "else" body have not been merged. This means the split due to the `c` branch is causing some differences in the two subtrees related to the target, and so removing the branch could make the PSS-CFG incorrect. Indeed, the `c` branch assigns different values to `flag` which ultimately causes different values to be assigned to the target `z`. Thus the branch must be kept to preserve the original program's semantics.

Finally, as a third step of our algorithm, we produce an equivalent C program from the PSS-CFG by "decompiling" it. The decompilation process is quite straightforward so we do not detail it here. It is done primarily so that external off-the-shelf applications can be executed on the PSS-CFG. The decompiled program for our example is shown in Fig. 1(d). At the outset, one can notice that the decompiled program has only 3 paths compared to 8 paths in the original program. Moreover, information that cannot be captured from the original program can be captured by the decompiled program. For instance, a concolic tester on the original program will always generate a value for `d` regardless of the value generated for `c`. However in the decompiled program, if the value of `c` was generated to be 0, the tester would not generate the value of `d` because it will not affect the target `z`. It can also be seen that the variable `flag`, which was mainly used for control flow between different parts of the code, is not even present in the decompiled program. This information cannot be captured by static slicers like [8, 15], which cannot statically remove the assignments to `flag` or the branch on `flag` from the program without becoming unsound.

Remark. One might wonder if our complete algorithm to produce the PSS-CFG is equivalent to simply expanding the paths of the original program producing a semantically equivalent program, deleting the infeasible paths, and applying standard slicing wrt the target. Even though conceptually it may be similar, there are many practical differences with our method. Without the merging performed by our algorithm, one would run into exponential blowup of paths during symbolic execution, before even producing the semantically equivalent program. Even if a merging mechanism is used to contain the blowup, without the guarantee of *lossless* merging provided by our algorithm, one could obtain imprecise dependency information thereby keeping irrelevant statements in the new program. However, our algorithm provides the right balance between precision and performance of such a target-based transformation. Thus, the process of constructing the SE tree and the process of dependency computation are closely intertwined and cannot be separated and outsourced to an external slicer.

4. 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, $\mathsf{assume}(\mathrm{c})$, 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. $\longrightarrow \subseteq \Sigma \times \Sigma \times \mathit{Ops}$ 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 $\mathsf{op} \in \mathit{Ops}$. We shall also use a similar notation $\upsilon \xrightarrow{\mathsf{op}} \upsilon'$ to denote a transition from the symbolic state υ to υ' corresponding to their program locations. 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. We use special symbols for initial location, $\ell_{\mathsf{start}} \in I$, and final location, $\ell_{\mathsf{end}} \in O$. The symbolic store s is a function from program variables to terms over input symbolic variables. The *evaluation* $[\![c]\!]_s$ of a constraint expression c in a store s is defined as: $[\![v]\!]_s = s(v)$ (if v is a variable), $[\![n]\!]_s = n$ (if n is an integer), $[\![e \ \mathsf{op} \ e']\!]_s = [\![e]\!]_s \ \mathsf{op} \ [\![e']\!]_s$ (where e, e' are expressions and op is a relational or arithmetic operator).

Finally, Π is called *path condition*, a first-order formula over the symbolic inputs that accumulates constraints which the inputs must satisfy in order to follow the 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 $\upsilon \equiv \langle \ell, s, \Pi \rangle \in \mathit{SymStates}$, the symbolic execution of $\ell \xrightarrow{\mathsf{op}} \ell'$ returns another symbolic state υ' defined as:

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

Note that Equation (1) queries a constraint solver for satisfiability checking on the path condition. We assume the solver is sound but not necessarily complete. That is, the solver 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 $[\![\upsilon]\!] : \mathit{SymStates} \rightarrow \mathit{FOL}$ 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 also define a (partial) function $\mathsf{MergePoint} : \mathit{SymStates} \rightarrow \mathit{SymStates} \times \mathit{SymStates}$ that, given a symbolic state $\upsilon \equiv \langle \ell, \cdot, \cdot \rangle$ if there is an assume statement at υ (i.e., ℓ corresponds to a branch point), returns a tuple $\langle \upsilon_1 \equiv \langle \ell', \cdot, \cdot \rangle, \upsilon_2 \equiv \langle \ell', \cdot, \cdot \rangle \rangle$ such υ_1 and υ_2 are reachable from υ, and ℓ' is the *nearest post-dominator* of ℓ. In other words, υ_1 and υ_2 are the symbolic states at the merge point reached through the "then" and "else" body respectively.

A *symbolic execution tree* contains all the execution paths explored during the symbolic execution of a transition system by triggering Equation (1). The nodes represent symbolic states and the arcs represent transitions between states.

Dependency computation via Abstract Interpretation. The backward dependency computation process starts from ℓ_{end} with a set of "target variables" $\mathcal{V} \subseteq \mathit{Vars}$, for which the program transformation is being performed. To compute the dependencies, we follow the dataflow approach described by Weiser [23] 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 $\sigma\upsilon \in \mathcal{D}$ is the approximate set of variables at the symbolic state υ that may affect variables in $\mathcal{V}$. *Backward data dependencies* can then be formulated as follows. Given a transition relation $\upsilon \xrightarrow{\mathsf{op}} \upsilon'$ we define *def*(op) and *use*(op) as the sets of variables written to and read during the execution of op, respectively. Then,

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

where $\sigma\upsilon' = \mathcal{V}$ if $\upsilon' \equiv \langle \ell_{\mathsf{end}}, \cdot, \cdot \rangle$. In the first case of Eqn. 2, we say that $\upsilon \xrightarrow{\mathsf{op}} \upsilon'$ is "included in the slice".

Backward control dependencies can also affect variables in $\mathcal{V}$. A transition relation $\delta \equiv \upsilon \xrightarrow{\mathsf{op}} \upsilon'$ where $\mathsf{op} \equiv \mathsf{assume}(\mathrm{c})$ is included in the slice if any transition relation from δ to its *nearest post-dominator* is included in the slice[3]. Then,

$$\sigma\upsilon \triangleq \sigma\upsilon' \cup \mathit{use}(\mathsf{op}) \quad (3)$$

Finally, a function $\widehat{pre}(\sigma\upsilon', \mathsf{op})$ that returns the *pre-state* $\sigma\upsilon$ after executing backwards the operation op with the *post-state* $\sigma\upsilon'$ is defined using Eqs. (2,3).

Tree transformation rules. The SE tree produced by our algorithm, together with the dependency information of each symbolic state, is represented using the set $\mathcal{S}$ of *facts* of the following types:

- $\mathsf{edge}(\upsilon \xrightarrow{\mathsf{op}} \upsilon')$, denoting a feasible edge from υ to υ'
- $\mathsf{inf\_edge}(\upsilon \xrightarrow{\mathsf{op}} \upsilon')$, denoting an infeasible edge from υ to υ'
- $\mathsf{merged}(\upsilon, \upsilon')$, denoting that υ has been merged with υ' (will be formalised later)
- $\mathsf{in\_slice}(\upsilon \xrightarrow{\mathsf{op}} \upsilon')$, denoting that the transition from υ to υ' is included in the slice due to Eqs. 2,3.

Note that we do not explicitly store the dependency information at each state, but rather just the fact whether a transition from the state is included in the slice or not (denoted by the $\mathsf{in\_slice}$ fact). In Section 5.2, the transformation of the SE tree into the final PSS-CFG will be modelled using certain rules that act upon these facts.

[3]We assume a function INFL (δ) that returns the set of transitions from δ to its nearest post-dominator – the set of transitions "influenced" by δ.

5. ALGORITHM

We describe our algorithm in two phases: in phase one (Section 5.1), we explore symbolic paths in the program to generate the symbolic execution (SE) tree annotated with dependencies. In phase two (Section 5.2), we transform this tree by removing edges and sub-trees, to finally produce the PSS-CFG.

At a high level, our algorithm performs forward symbolic execution in a depth-first manner interleaved with backward dependency computation. Symbolic execution avoids the exploration of infeasible paths, thus increasing the precision of the computed dependencies. However, it allows multiple copies of the same program point to exist as different *symbolic states*, along different symbolic paths. Thus an important challenge to overcome is to avoid this inherent exponential blowup of symbolic execution.

Our solution is to merge different symbolic states provided certain conditions are met. These "merging conditions" guarantee that the merge does not incur any loss of slicing information. To formalise these conditions, we define two key concepts:

DEFINITION 1 (INTERPOLANT). *Given a pair of first order logic formulas A and B such that $A \wedge B$ is $false$, an* interpolant *[7] is another 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 has been prominently used to reduce state space blowup in program verification [20, 16] and testing [13]. Here we use it for a similar purpose – to merge states and thereby avoid redundant exploration. However, in addition to merging states, we must also guarantee *lossless merging*. Thus, we define:

DEFINITION 2 (WITNESS PATHS & FORMULAS). *Given a symbolic state $\upsilon \equiv \langle \ell, \cdot, \cdot \rangle$ annotated with the set of dependency variables σ_υ, a* witness path *for a variable $v \in \sigma_\upsilon$ is a feasible symbolic path $\pi \equiv \upsilon \cdot \ldots \cdot \upsilon_{\mathsf{end}}$ such that $\upsilon_{\mathsf{end}} \equiv \langle \ell_{end}, \cdot, \cdot \rangle$ and there exists $v_1 \in \mathcal{V}$ such that v_1 is control- or data-dependent on v along the path π. We call $[\![\upsilon_{\mathsf{end}}]\!]$ the* witness formula *of v, denoted ω_υ.*

Intuitively, a witness path for a dependency variable at a symbolic state is a path arising from it along which the dependency variable affects the target variables at the end. The witness formula is the path condition of its witness path.

To accommodate witness formulas in our abstract domain $\mathcal{D}$ we redefine it as follows: $\mathcal{D} \triangleq \{\bot\} \cup \mathcal{P}(Vars \times FOL)$, 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. The abstract operations $\sqcup$ and $\sqsubseteq$ still stand for $\cup$ and $\subseteq$, but the pre-operator $\widehat{pre}$ is slightly modified to propagate back witness formulas: the dependency variable x is still computed using Eqs. 2,3 as before, while the pre-state witness formula for x is the computed as the conjunction of the post-state witness formula for x with the logical constraint from the transition operation op. We now formalise our merging conditions:

DEFINITION 3 (MERGING CONDITIONS). *Given a current symbolic state $\upsilon \equiv \langle \ell, s, \Pi \rangle$ and an already annotated symbolic state $\upsilon' \equiv \langle \ell, s', \Pi' \rangle$ such that $\overline{\Psi}_{\upsilon'}$ is the interpolant generated for υ' and $\sigma_{\upsilon'}$ are the dependencies at υ', we say υ is* merged *with υ' if the following conditions hold:*

$$\begin{array}{ll} (a) & [\![\upsilon]\!] \models \overline{\Psi}_{\upsilon'} \\ (b) & \forall \langle x, \cdot \rangle \in \sigma_{\upsilon'} \bullet \exists \langle x, \omega_x \rangle \in \sigma_{\upsilon'} \; s.t \\ & [\![\upsilon]\!] \wedge \omega_x \text{ is satisfiable} \end{array} \quad (4)$$

Note importantly that both υ and υ' must correspond to the same program point ℓ in order to be merged. The condition (a) affects *soundness* and it ensures that the set of feasible symbolic paths reachable from υ is a subset of those from υ'. This is a necessary condition for two states to be merged.

LEMMA 1. *Given states $\upsilon \equiv \langle \ell, s, \Pi \rangle$ and $\upsilon' \equiv \langle \ell, s', \Pi' \rangle$, let $\overline{\Psi}_{\upsilon'}$ be the interpolant for υ'. If $[\![\upsilon]\!] \models \overline{\Psi}_{\upsilon'}$, the set of feasible paths from υ is a subset of those from υ'.*

PROOF. *(By contradiction). Assume there exists a feasible path π, with path condition Π_π, from υ but π is infeasible from υ'. If π is infeasible from υ' then $[\![\upsilon']\!] \wedge \Pi_\pi$ is unsatisfiable, and by definition of interpolant, $\overline{\Psi}_{\upsilon'} \wedge \Pi_\pi$ is unsatisfiable. Since $[\![\upsilon]\!] \models \overline{\Psi}_{\upsilon'}$, it follows that $[\![\upsilon]\!] \wedge \Pi_\pi$ is unsatisfiable. However, since π is feasible from υ, $[\![\upsilon]\!] \wedge \Pi_\pi$ cannot be unsatisfiable.* □

The intuition is that the interpolant $\overline{\Psi}_{\upsilon'}$ represents the reason of infeasibility of all infeasible paths arising from υ'. If $[\![\upsilon]\!] \models \overline{\Psi}_{\upsilon'}$, any infeasible path under υ' is also infeasible under υ. In other words, any feasible path under υ is also feasible under 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 merging of two states does not incur any loss of precision. This is formalised as follows.

THEOREM 1. *Given states $\upsilon \equiv \langle \ell, s, \Pi \rangle$ and $\upsilon' \equiv \langle \ell, s', \Pi' \rangle$, let $\sigma_{\upsilon'}$ be the dependencies and witness formulas associated with υ'. If υ can be merged with υ' then by exploring υ there cannot be produced a set of dependencies σ_υ such that $\sigma_\upsilon \neq \sigma_{\upsilon'}$.*

PROOF. *Assume that although υ can be merged with υ' (i.e., both conditions of Eqn. 4 are satisfied), it is instead symbolically explored and a dependency set σ_υ is obtained.*

***Proof that** $\sigma_{\upsilon'} \subseteq \sigma_\upsilon$**:** Since υ can be merged with υ', by condition (b) of Eqn. 4, $\forall \langle x, \cdot \rangle \in \sigma_{\upsilon'}$, there is a witness path, say π_x, with formula ω_x such that $[\![\upsilon]\!] \wedge \omega_x$ is satisfiable. That is, π_x is feasible from υ. By the definition of a witness path (Definition 2), $\exists v_1 \in \mathcal{V}$ s.t v_1 is control- or data-dependent on x along the path π_x, which is feasible from υ. Therefore x must be in σ_υ.*

***Proof that** $\sigma_\upsilon \subseteq \sigma_{\upsilon'}$**:** (by contradiction) Assume $\exists x \in \sigma_\upsilon$ s.t $x \notin \sigma_{\upsilon'}$. Then, the witness path for x, π_x with formula ω_x must be such that $[\![\upsilon]\!] \wedge \omega_x$ is satisfiable but $[\![\upsilon']\!] \wedge \omega_x$ is unsatisfiable (otherwise from the definition of a witness path, x would have been included in $\sigma_{\upsilon'}$). That is, π_x is feasible from υ but infeasible from υ'. From Eqn. 4 condition (a) and Lemma 1, this is impossible.* □

5.1 Generating the SE Tree Structure with Dependencies

The purpose of our main algorithm, GENPSSCFG (Fig. 2), is to generate a finite symbolic execution tree annotated with dependency information at each symbolic state. As mentioned in Section 4, the tree is represented using a set of facts that are added to the set $\mathcal{S}$, which is assumed to be a global variable to the algorithm. GENPSSCFG requires the program to have been translated to a transition system $\langle \Sigma, I, \longrightarrow, O \rangle$ in SSA form, and accepts a symbolic state as argument. It is initiated with the state $\upsilon \equiv \langle \ell_{\mathsf{start}}, \epsilon, true \rangle$. GENPSSCFG implements a mutually recursive algorithm with a few other procedures.

First, the most important decision of whether to merge a symbolic state with another is taken by GENPSSCFG at line 1. It attempts to find another symbolic state υ' such that υ and υ' satisfy the two merging conditions in Equation 4. If yes, it merges υ with υ' by calling the procedure MERGE at line 2. If such a υ' does not exist, GENPSSCFG decides whether to split the symbolic execution of υ or not by checking if υ corresponds to a branching

GENPSSCFG ($\upsilon \equiv \langle \ell, s, \Pi \rangle$)
1: **if** $\exists\, \upsilon' \equiv \langle \ell, s', \Pi' \rangle$ s.t. υ and υ' satisfy Eqn. 4
2: **then** MERGE (υ, υ')
3: **else if** υ is at a branch point **then**
4: SPLIT(υ)
5: **else**
6: SYMEXEC (υ)

MERGE (υ, υ')
1: $\overline{\Psi}_\upsilon := \overline{\Psi}_{\upsilon'}$
2: $\sigma_\upsilon := \sigma_{\upsilon'}$
3: $\mathcal{S}$:=$\mathcal{S} \cup$ merged(υ, υ')

SPLIT ($\upsilon \equiv \langle \ell, s, \Pi \rangle$)
1: $\overline{\Psi}_\upsilon$:= true
2: **foreach** transition $\ell \xrightarrow{\text{assume(c)}} \ell'$ **do**
3: **if** (υ is a loop header) **then**
4: $\upsilon' \triangleq \langle \ell', \cdot, \text{invariant}(\upsilon) \wedge [\![\text{c}]\!]s \rangle$
5: **else**
6: $\upsilon' \triangleq \langle \ell', s, \Pi \wedge [\![\text{c}]\!]s \rangle$
7: **if** υ' is infeasible state **then**
8: $\mathcal{S}$:=$\mathcal{S} \cup$ inf_edge$(\upsilon \xrightarrow{\text{assume(c)}} \upsilon')$
9: $\overline{\Psi}_{\upsilon'}$:= false, $\sigma_{\upsilon'} := \emptyset$
10: **else**
11: $\mathcal{S}$:=$\mathcal{S} \cup$ edge$(\upsilon \xrightarrow{\text{assume(c)}} \upsilon')$
12: GENPSSCFG (υ')
13: $\overline{\Psi}_\upsilon := \overline{\Psi}_\upsilon \wedge \widehat{wlp}\,(\overline{\Psi}_{\upsilon'}, \text{assume(c)})$
14: $\sigma_\upsilon := \sigma_\upsilon \sqcup \widehat{pre}\,(\sigma_{\upsilon'}, \text{assume(c)}, s)$
15: **if** $\delta \equiv \upsilon \xrightarrow{\text{assume(c)}} \upsilon'$ satisfies Eqn. 3 **then**
16: $\mathcal{S}$:=$\mathcal{S} \cup$ in_slice$(\upsilon \xrightarrow{\text{assume(c)}} \upsilon')$

SYMEXEC ($\upsilon \equiv \langle \ell, s, \Pi \rangle$)
1: **if** $\nexists$ transition relation $\ell \xrightarrow{\text{x:=e}} \ell'$ **then**
2: $\overline{\Psi}_\upsilon$:=true, $\sigma_\upsilon := \mathcal{V}$
3: **else**
4: $\upsilon' \triangleq \langle \ell', s[x \mapsto [\![e]\!]s], \Pi \rangle$
5: $\mathcal{S}$:=$\mathcal{S} \cup$ edge$(\upsilon \xrightarrow{\text{x:=e}} \upsilon')$
6: **if** υ' is not a loop header
7: GENPSSCFG (υ')
8: $\overline{\Psi}_\upsilon := \widehat{wlp}\,(\overline{\Psi}_{\upsilon'}, \text{x:=e})$
9: $\sigma_\upsilon := \widehat{pre}\,(\sigma_{\upsilon'}, \text{x:=e})$
10: **if** $\upsilon \xrightarrow{\text{x:=e}} \upsilon'$ satisfies Eqn. 2 **then**
11: $\mathcal{S}$:=$\mathcal{S} \cup$ in_slice$(\upsilon \xrightarrow{\text{x:=e}} \upsilon')$

Figure 2: Symbolic execution interleaved with dependency computation to produce the SE tree

point in the program (line 3). If yes it calls the procedure SPLIT at line 4 which, as we will see, forks the symbolic execution of different branches from υ. If both the above cases do not match, GENPSSCFG simply continues the symbolic execution by calling the procedure SYMEXEC with υ. GENPSSCFG is in essence the high level backbone of our method.

The procedure MERGE, given a current symbolic state υ and an already explored state υ', merges the former with the latter by setting the interpolant and dependency set of υ to those of υ'. Recall that Theorem 1 guaranteed such a merge to have no loss of precision. That is, had υ been explored instead of being merged with υ', the resulting dependency set at υ would be exactly $\sigma_{\upsilon'}$. Finally the procedure adds the fact merged(υ, υ') to $\mathcal{S}$ to record the merge between the two states.

The procedure SPLIT is used to fork the symbolic execution of a state from which multiple transitions are possible (typically a branch point). Given a symbolic state υ with program point ℓ and path condition Π, it first initialises its interpolant $\overline{\Psi}\upsilon$ to true at line 1. At line 2 it iterates its main body over each transition possible from υ. Now there is an issue: if the current state is a loop header (line 4), then symbolically executing the loop could result in an *unbounded* tree, which we want to avoid. Therefore, we need to execute the loop with a *loop invariant* to make the tree finite.

Our method to compute a loop invariant is simple but effective: from the loop header's symbolic state υ, we only keep the constraints that are unchanged through the loop, and delete the rest. For instance, if $x > 5$ holds at the loop header and x is only incremented in the loop, then $x > 5$ is unchanged through the loop. This widened state at υ ultimately forms a loop invariant. This technique provides a balance between getting the strongest invariant – which is needed to maximise path-sensitivity – and efficiency. We found experimentally that this technique preserves most of the important information through the loop. Nevertheless, we remind the reader that no matter what the invariant is, it does not affect the guarantee of lossless-ness of dependency information during our merging, and the correctness of our transformation as stated by Theorem 2.

We assume a function invariant that given a symbolic state υ, returns a FOL formula representing the loop invariant. With this invariant, the next state is constructed by augmenting it with $[\![\text{c}]\!]s$ where c is the branching condition of the assume statement (line 4). If not, υ' is constructed (line 6) by augmenting the path condition Π with $[\![\text{c}]\!]s$. At line 7 an important check is performed: if υ' is an infeasible state (i.e., the augmented path condition is unsatisfiable), it means symbolic execution has encountered an infeasible path. Therefore it adds to $\mathcal{S}$ the fact that the transition from υ to υ' is infeasible (line 8), and sets the interpolant and dependency set of υ' to false and $\emptyset$ respectively (line 9) to signify that the state is unreachable. Otherwise it adds a normal edge to $\mathcal{S}$ at line 11 and (mutually) recursively calls GENPSSCFG with υ'.

In either case, υ' would have been annotated with an interpolant $\overline{\Psi}_{\upsilon'}$ and dependency set $\sigma_{\upsilon'}$. Now it computes the same information for υ at lines 13-14. The interpolant $\overline{\Psi}_\upsilon$ is supposed to generalise the SE tree below υ while preserving its infeasible paths. For this, the procedure $\widehat{wlp}$: *FOL* $\times$ *Ops* $\rightarrow$ *FOL* is called that ideally computes the *weakest liberal precondition* [9], the weakest formula on the initial state ensuring the execution of assume(c) results in the state $\overline{\Psi}_{\upsilon'}$. In practice we approximate *wlp* by making a linear number of calls to a theorem prover following techniques described in [16], usually resulting in a formula stronger than the weakest liberal precondition. The dependency set σ_υ is computed

RULE 1 (STRAIGHT LINE SLICING)

$$\frac{E_1 \equiv \mathsf{edge}(v_0 \xrightarrow{\mathsf{op}} v_1) \in \mathcal{S} \qquad E_2 \equiv \mathsf{edge}(v_1 \xrightarrow{\mathsf{x:=e}} v_2) \in \mathcal{S} \qquad \mathsf{in\_slice}(v_1 \xrightarrow{\mathsf{x:=e}} v_2) \notin \mathcal{S}}{\mathcal{S} := \mathcal{S} \setminus \{E_1, E_2\} \cup \{\mathsf{edge}(v_0 \xrightarrow{\mathsf{op}} v_2)\}}$$

RULE 2 (INFEASIBLE PATH REMOVAL)

$$\frac{E_1 \equiv \mathsf{edge}(v_0 \xrightarrow{\mathsf{op}} v_1) \in \mathcal{S} \qquad E_2 \equiv \mathsf{edge}(v_1 \xrightarrow{\mathsf{assume(c_1)}} v_2) \in \mathcal{S} \qquad E_3 \equiv \mathsf{inf\_edge}(v_1 \xrightarrow{\mathsf{assume(c_2)}} v_3) \in \mathcal{S}}{\mathcal{S} := \mathcal{S} \setminus \{E_1, E_2, E_3\} \cup \{\mathsf{edge}(v_0 \xrightarrow{\mathsf{op}} v_2)\}}$$

RULE 3 (TREE SLICING)

$$\frac{\begin{array}{c}\mathsf{edge}(v_0 \xrightarrow{\mathsf{op}} v_1) \in \mathcal{S} \qquad \mathsf{edge}(v_1 \xrightarrow{\mathsf{assume(c_1)}} v_2) \in \mathcal{S} \qquad \mathsf{edge}(v_1 \xrightarrow{\mathsf{assume(c_2)}} v_3) \in \mathcal{S} \qquad v_2 \neq v_3 \\ \mathsf{in\_slice}(v_1 \xrightarrow{\mathsf{assume(c_1)}} v_2) \notin \mathcal{S} \qquad \mathsf{in\_slice}(v_1 \xrightarrow{\mathsf{assume(c_2)}} v_3) \notin \mathcal{S} \qquad \langle v_k, v_k' \rangle \equiv \mathsf{MergePoint}(v_1) \qquad \mathsf{merged}(v_k, v_k') \in \mathcal{S}\end{array}}{\mathcal{S} := \mathcal{S} \setminus \{\mathsf{edge}(v' \xrightarrow{\mathsf{op}} v'') \mid v' \xrightarrow{\mathsf{op}} v'' \in \mathsf{INFL}(v_1 \xrightarrow{\mathsf{assume(c_1)}} v_2) \vee v' \xrightarrow{\mathsf{op}} v'' \in \mathsf{INFL}(v_1 \xrightarrow{\mathsf{assume(c_2)}} v_3)\} \cup \{\mathsf{edge}(v_0 \xrightarrow{\mathsf{op}} v_k)\}}$$

Figure 3: Transformation rules to produce the final PSS-CFG

by applying the pre-operation $\widehat{pre}$ on $\sigma_{v'}$ and joining with any existing set (across different iterations of the main loop).

Finally, in lines 15-16 of SPLIT, it checks if any transition from δ to its nearest postdominator is included in the slice (Eqn. 3). If yes, it adds an in_slice fact to $\mathcal{S}$ with the transition from v to v'.

The final procedure SYMEXEC is called by GENPSSCFG when the current symbolic state v corresponding to program point ℓ cannot split (typically an assignment statement). Initially, at line 1, it checks if there exists a program transition from ℓ to any other ℓ'. If not, symbolic execution has reached the end of a (feasible) path whose final state is v. In other words, it has reached a terminal node. Hence it sets the interpolant $\overline{\Psi}_v$ to true and its dependency set σ_v to $\mathcal{V}$ (recall that the target variables are specified at ℓ_{end}) at line 2.

If there exists a transition from ℓ to say ℓ' with the assignment x:=e, it constructs the next symbolic state (line 4) v' by setting in the store s the value of x to $[\![\mathsf{e}]\!]s$ and adds to $\mathcal{S}$ the appropriate edge fact (line 5). Then, if v' is not a loop header, it recursively calls GENPSSCFG with v' (line 7). If v' is a loop header, then there is no need to explore it again since it would have already been explored with the loop invariant (at SPLIT line 4). Our algorithm thus makes the symbolic execution finite. In SYMEXEC line 8 and 9, it sets the interpolant (and dependency set) of v by calling $\widehat{wlp}$ (and $\widehat{pre}$) on the interpolant (and dependency set) of v'. Finally, at lines 10-11, if x contains a variable in $\sigma_{v'}$ (Eqn. 2) it adds to $\mathcal{S}$ the fact that the transition from v to v' is included in the slice.

To perform the fixpoint computation at the highest level, we keep making calls to GENPSSCFG until there is no change in $\mathcal{S}$. This is the simplest way to describe the fixpoint computation but in practice we can optimise it by calling GENPSSCFG with the symbolic state of the loop header in which the change was detected.

5.2 Transformation of the Annotated SE Tree

The algorithm described so far produces a symbolic execution tree represented as a set of facts $\mathcal{S}$. Now we present certain rules in Fig. 3 that act upon $\mathcal{S}$ to modify it, in essence modelling the transformation of the SE tree into the final PSS-CFG. The rules are presented in a declarative fashion and can be implemented conveniently in a rule-based programming language (e.g., Constraint Handling Rules).

STRAIGHT LINE SLICING states that if there is a transition (or edge) from state v_0 to v_1 and an *assignment* transition from v_1 to v_2 such that the latter is not included in the slice, then both transitions can be removed and replaced with one linking v_0 directly to v_2. This is the typical rule for slicing assignment statements using dependencies.

INFEASIBLE PATH REMOVAL states that if there is a transition from state v_0 to v_1, and v_1 is a *branch point* such that there is branching edge (edge) from v_1 to v_2 and an infeasible branching edge (inf_edge) from v_1 to another v_3, then all three edges can be removed and v_0 can be directly linked to the feasible state v_2.

TREE SLICING is more complicated and the most powerful in terms of reducing the symbolic state space of the PSS-CFG. It states that if there is a transition from v_0 to v_1, and v_1 is a branching point with branching transitions to v_2 (with condition assume(c$_1$)) and v_3 (with condition assume(c$_2$)) such that neither transition is included in the slice, then we can remove all transitions $v' \xrightarrow{\mathsf{op}} v''$ that occur either in the dynamic range of influence (given by INFL) of $v_1 \longrightarrow v_2$ or $v_1 \longrightarrow v_3$. In other words, we can remove all transitions that occur in the "then" or "else" body of the branch at v_1. But there is a problem: since we are working on a symbolic tree, removing the branch point v_1 would conceptually leave two different subtrees "hanging" without a parent. The question arises as to which subtree should we link to the node v_0. TREE SLICING guarantees that if the symbolic states at the end of the branch $\langle v_k, v_k' \rangle$ (as returned by MergePoint(v_1)) are merged by our algorithm (i.e., merged(v_k, v_k') exists), the differences in the trees do not affect the target variables. Hence it simply adds a transition directly linking v_0 to one of the symbolic states v_k.

We explain the reasoning behind the above rules by defining our correctness statement for the transformation of the SE tree into the PSS-CFG and providing a proof outline for it. First let two CFGs be defined *equivalent* wrt target variables $\mathcal{V}$ if for any input, the programs corresponding to both CFGs produce the same values for all variables in $\mathcal{V}$.

THEOREM 2. *(Correctness of transformation) An application of* RULE 1*,* RULE 2 *or* RULE 3 *to a CFG G produces a transformed CFG G' such that G' is equivalent to G wrt target variables $\mathcal{V}$.*

PROOF OUTLINE. The correctness of STRAIGHT LINE SLICING follows directly from the correctness of slicing assignment statements using dependency information, formalised in Eqn. 2. As for INFEASIBLE PATH REMOVAL, for any input that executes a path in G leading to the state v_1, the condition c_1 will evaluate to true

Benchmark	Lines of code			Blow	PSS	#Rule Triggers		
	Orig	St.slice	PSS	up	Time	Rul1	Rul2	Rul3
cdaudio	1817	1599	4452	2.78	24s	2685	1101	169
diskperf	937	706	2967	4.20	18s	1594	1132	73
floppy	1005	766	2086	2.72	7s	1062	651	99
floppy2	1513	1250	3507	2.81	16s	1514	819	120
kbfiltr	549	275	170	0.62	1s	111	46	7
kbfiltr2	782	492	410	0.83	1s	249	69	23
tcas	286	227	311	1.37	2s	138	204	47

(a)

Testing Time		Speed	#Solver calls	
St.slice	PSS	up	St.slice	PSS
1m30s	43s	2.1	16k	7k
900m	34m	26.5	26mil	1mil
9m6s	24s	22.8	260k	4k
525m	429m	1.2	613k	479k
2s	1s	2	63	52
22s	6s	3.7	7k	2k
4s	1s	4	1.5k	188
23h56m	**7h44m**	**3.1**	**26.9mil**	**1.5mil**

(b)

Table 1: (a) Statistics about the PSS-CFG (b) Experiments on the PSS-CFG for concolic testing

and c_2 will evaluate to false. Moreover, an assume statement does not modify any variable in the program state. Thus, both checks assume(c_1) and assume(c_2) are useless because we deterministically know their outcomes, and hence can be replaced with a transition linking v_0 to the next feasible state v_2 to produce G'.

The correctness proof of TREE SLICING is as follows. Assume that some input executes a path in G starting from v_{start} to v_0 and then reaches v_1. W.l.o.g, assume that the condition c_1 holds at v_1, therefore it chooses to follow v_2, reaches the merged point v_k and continues to eventually reach the terminal state v_{end}. Let us call this executed path π_G. In G', obtained by applying TREE SLICING on G, thereby removing the entire branch at v_1, the same input would follow a path, say $\pi_{G'}$, such that $\pi_{G'}$ is the exact same path as π_G starting from v_{start} till v_0, thus having the same symbolic state at v_0. At this point, $\pi_{G'}$ differs from π_G by implicitly "skipping" the execution of the branch at v_1 and instead directly reaches v_k.

Since v_k and v'_k were merged, the dependency sets at both points are the same. Now, since the transition $v_1 \xrightarrow{\text{assume}(c_1)} v_2$ in G was not included in the slice, it means that no statement "skipped" by $\pi_{G'}$ affected the dependency information at v_k. This implies that the symbolic state of the path $\pi_{G'}$ at v_k is the same as the symbolic state of the path π_G at v_k *as far as the dependency variables at* v_k *are concerned*. To be precise, the values of the dependency variables at v_k are the same in both π_G and $\pi_{G'}$. Since these are the only variables affecting the target variables $\mathcal{V}$ at v_{end}, it is sufficient to preserve their values to ensure that $\pi_{G'}$ will produce the same values for $\mathcal{V}$ as π_G. Of course $\pi_{G'}$ may produce different values than π_G for variables *not* in $\mathcal{V}$, but we are not interested in those variables. □

The three rules are applied until fixpoint is reached (i.e., none of them can be applied anymore). Termination of rule applications is guaranteed from the initial finiteness of the set $\mathcal{S}$ and the fact that all three rules remove more edges from $\mathcal{S}$ than they add. Soundness of individual rule applications is guaranteed from Theorem 2. Transitiveness of the rules is also guaranteed by Theorem 2 since each new CFG is equivalent to the previous CFG. Once fixpoint is reached, the final PSS-CFG structure can be extracted from $\mathcal{S}$.

Thus, Theorem 2 guarantees that the PSS-CFG is equivalent to the original program wrt the target variables $\mathcal{V}$. Therefore, any analysis of the original program concerned only with $\mathcal{V}$ can be applied on the PSS-CFG instead to take advantage of its benefits. We will see two such applications: program testing and verification.

6. EXPERIMENTAL EVALUATION

We evaluate the PSS-CFG using applications of program testing and verification to show considerable increase in their performance. We implemented the algorithm on the TRACER [14] framework for symbolic execution. Our proof-of-concept implementation models the heap as an array. A flow-insensitive pointer analysis provided by Crystal [22] is used to partition updates and reads into alias classes where each class is modelled by a different array. 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 to by q. This coarse modelling of heaps does introduce imprecision in the analysis, but it is orthogonal to our main contribution. Functions are inlined during symbolic execution and external functions are modelled as having no side effects and returning an unknown value.

We used device drivers from the ntdrivers-simplified category of SV-COMP 2013 [4] and a traffic collision avoidance program called tcas as benchmarks, and chose the target variables from the safety properties of the programs. All programs had multiple target safety properties on several variables, all of which were included in our slicing criteria. For practically applying external tools on the PSS-CFG structure, we used its equivalent decompiled program. Since both the original and decompiled programs are in C, we can easily measure how external tools benefit from our transformation.

For all our experiments we compare the PSS-CFG[4] with a static slice of the benchmark program on the target variables. Comparing with a static slice is more challenging as some statements would have already been sliced away from the original program. We obtained the static slice through the well-known state-of-the-art slicer Frama-C [8, 1]. Frama-C is a path-sensitive static slicer that can detect infeasible paths through techniques such as constant propagation, constant folding and abstract interpretation. Also, before the target variables are provided and our algorithm is initiated, we process the program and store an intermediate representation (IR). This processing involves computing information about infeasible paths in the program and is completely independent of the target variables. Then, when the target variables are provided, our algorithm is invoked and it uses information from this IR. All experiments were run on an Intel 3.2 Ghz system with 2GB memory.

Now, we provide statistics about the PSS-CFG and its construction in Table 1(a). The Lines of code column shows the number of non-commented lines of code in the original (Orig) program, its static slice (St.slice) and its decompiled program (PSS) respectively. In the column Blowup we show the ratio of the LOC of PSS-CFG compared to the static slice. The blowup is a result of the balance between the splits introduced by path-sensitivity, and the merges and slicing from our algorithm. It is clear that the blowup is manageable, sometimes even smaller than the program, being on average around 2. In the column PSS Time we show the time

[4]We use "decompiled program" and "PSS-CFG" interchangeably.

Benchmark	IMPACT			ARMC			CPA-CHECKER		
	Verification Time		Speed	Verification Time		Speed	Verification Time		Speed
	St.Slice	PSS	up	St.Slice	PSS	up	St.Slice	PSS	up
cdaudio	95s	14s	6.8	T/O	21s	N/A	26s	14s	1.86
diskperf	146s	18s	8.1	T/O	6s	N/A	7s	6s	1.17
floppy	34s	8s	4.3	259s	6s	43.17	6s	5s	1.20
floppy2	39s	13s	3.0	T/O	17s	N/A	10s	8s	1.25
kbfiltr	4s	1s	4.0	3s	1s	3.00	3s	2s	1.50
kbfiltr2	8s	2s	4.0	13s	2s	6.50	4s	2s	2.00
tcas	3s	1s	3.0	3s	1s	3.00	2s	1s	2.00
Total	**329s**	**57s**	**5.8**	**T/O**	**54s**	**N/A**	**58s**	**38s**	**1.53**
	(a)			(b)			(c)		

Table 2: Experiments on the quality of PSS-CFG for verification times of different verifiers

taken in seconds for our algorithm to produce the PSS-CFG given the target variables, which is modest. In the final column #Rule Triggers we show the number of times each transformation rule was triggered during PSS-CFG construction. Although RULE 3 is shown to be triggered fewer number of times than RULE 1 or RULE 2, it is the most powerful rule in reducing the search space of the PSS-CFG. In tcas we see RULE 2 triggering more frequently than RULE1 due to its large number of infeasible paths.

Note that the PSS-CFG construction is only performed once for a given set of target variables. The resulting program can however be subjected to an innumerable number of properties to be verified or tested. For example, using the same PSS-CFG, one can verify different bounds on a target variable depending on different preconditions to the program.

6.1 Testing (white-box)

We consider software testing an important application for the PSS-CFG to be used. For this, we consider the typical DART [10] methodology that performs *concolic testing*, i.e., executing the program with both concrete and symbolic inputs and symbolically negating branches to explore new paths. We chose the publicly available concolic tester CREST, an implementation of DART for C programs. Since the statically sliced and decompiled programs are in C, the experiment was simply to run the concolic testing process on both programs and measure the time taken to complete, i.e., time taken to test all feasible paths in the program.

In Table 1(b), we show the measures of the experiment. The second and third columns (St.slice and PSS) show the time taken to complete the concolic testing process on the statically sliced and decompiled programs respectively. The third column shows the Speedup obtained by using the PSS-CFG, i.e., the ratio of the columns St.slice and PSS. It is immediately apparent that the PSS-CFG provides speedup in all benchmarks. In programs diskperf and floppy the speedup is exceptionally high around 22-26, reducing the concolic testing time from, for instance, 900 minutes (15 hours) to just 34 minutes. On the other hand, in floppy2 the speedup of 1.2 is not that high, but still the absolute benefit in time can be seen – around 96 minutes or 1.5 hours. Ultimately, the total time taken for concolic testing to run on all our statically sliced programs was almost 24 hours, whereas it took less than 8 hours to run on the decompiled programs, providing a net benefit in time of a magnitude of 3.1. Although it is understood that in practice concolic testing may not terminate by exploring all paths, we gave a huge timeout (24 hours) for the process to terminate simply to see how much benefit the PSS-CFG can provide in timing. From the table, it is clear that the PSS-CFG can make the difference between termination and timing-out of the concolic testing process.

In addition to time, we also measured the number of calls made by CREST to its underlying solver. This measure, shown in the column #Solver calls, gives an idea of how the PSS-CFG would still benefit the concolic tester even if a different, faster solver was used. Again we see several magnitudes of less solver calls for all benchmarks when CREST was run on the PSS-CFG. The maximum benefit is in diskperf where 26 million calls were made for the statically sliced program, compared to only 1 million for the decompiled program. This is in-line with the speedup in time for diskperf, around 26. This indicates that even if a faster solver is used, the relative speedup in time for this benchmark would still be around 26, although the absolute timings may be faster. Ultimately, this table shows that concolic testing would definitely benefit by using the PSS-CFG instead of the statically sliced program.

6.2 Verification

Another important application for the PSS-CFG is program verification. In Table 2 we compare the verification times of the benchmarks across three different state-of-the-art verifiers: IMPACT [19], ARMC [21] and CPA-CHECKER [5]. We chose this set of verifiers because they come from different approaches to verification – interpolant-based, CEGAR-based and SMT-based. Since IMPACT is not publicly available, we use CPA-CHECKER's implementation of the IMPACT algorithm. In each table, the second and third columns show the verification time (in seconds) of the statically sliced program (St.Slice) and the PSS-CFG (PSS), respectively. In the third column Speedup, we show the ratio of St.slice to PSS.

For all three verifiers, it can be clearly seen that the PSS-CFG is verified in a much faster time than the static slice. For IMPACT, verifying all statically sliced programs in our suite took 329 seconds whereas verifying the respective PSS-CFGs took only 57 seconds. Thus, the speedup across all programs on aggregate is 5.8. As for ARMC, it was unable to terminate its verification of the statically sliced programs for cdaudio, diskperf and floppy2 with a timeout of 10 minutes, whereas it was able to verify each of their respective PSS-CFGs in less than 30 seconds, thus providing a huge benefit to ARMC. For CPA-CHECKER, the benefit was relatively smaller, providing on average a speedup of 1.5. The reason is because CPA-CHECKER is a more sophisticated verifier than the other two, but still the fact that the PSS-CFG provides a speedup for CPA-CHECKER is to be considered noteworthy. Thus, we believe that the PSS-CFG is quite a useful object in general for verification.

7. ACKNOWLEDGEMENT

We would like to thank Jorge Navas for his contributions to this work while at NUS.

8. REFERENCES

[1] Frama-C Software Analyzers. `http://frama-c.com/`.

[2] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. ICSE, 2014.

[3] G. Balakrishnan, S. Sankaranarayanan, F. Ivancic, O. Wei, and A. Gupta. SLR: Path-Sensitive Analysis through Infeasible-path Detection and Syntactic Language Refinement. In *SAS*, 2008.

[4] D. Beyer. Second Competition on Software Verification. In *TACAS*, 2013.

[5] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *CAV*, 2011.

[6] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, pages 351–366, 2008.

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

[8] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, 2012.

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

[10] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.

[11] M. Harman and S. Danicic. Amorphous Program Slicing. WPC, 1997.

[12] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing Software for Model Construction. *Higher Order Symbol. Comput.*, 2000.

[13] J. Jaffar, V. Murali, and J. Navas. Boosting Concolic Testing via Interpolation. In *FSE*, 2013.

[14] J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV 2012*, pages 758–766, 2012.

[15] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Path-Sensitive Backward Slicing. In *SAS*, pages 231–247, 2012.

[16] J. Jaffar, A. E. Santosa, and R. Voicu. An Interpolation Method for CLP Traversal. In *CP*, 09.

[17] R. Jhala and R. Majumdar. Path Slicing. In *PLDI*, 2005.

[18] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *PLDI*, 2012.

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

[20] K. L. McMillan. Lazy Annotation for Program Testing and Verification. In T. Touili, B. Cook, and P. Jackson, editors, *22nd CAV*, volume 6174 of *LNCS*, pages 104–118. Springer, 2010.

[21] A. Podelski and A. Rybalchenko. ARMC. In *PADL'07*.

[22] R. Rugina, M. Orlovich, and X. Zheng. Crystal: A Program Analysis System for C. http://www.cs.cornell.edu/projects/crystal, 2007. [Online; accessed 09-July-2011].

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