Program Analysis via Satisfiability Modulo Path Programs

William R. Harris
University of Wisconsin, Madison, WI.
wrharris@cs.wisc.edu

Sriram Sankaranarayanan
University of Colorado, Boulder, CO.
srirams@colorado.edu

Franjo Ivančić Aarti Gupta
NEC Laboratories America, Princeton.
{ivancic,agupta}@nec-labs.com

Abstract

Path-sensitivity is often a crucial requirement for verifying safety properties of programs. As it is infeasible to enumerate and analyze each path individually, analyses compromise by soundly merging information about executions along multiple paths. However, this frequently results in a loss of precision. We present a program analysis technique that we call *Satisfiability Modulo Path Programs* (SMPP), based on a path-based decomposition of a program. It is inspired by insights that have driven the development of modern SMT (Satisfiability Modulo Theory) solvers. SMPP symbolically enumerates path programs using a SAT formula over control edges in the program. Each enumerated path program is verified using an oracle, such as abstract interpretation or symbolic execution, to either find a proof of correctness or report a potential violation. If a proof is found, then SMPP extracts a sufficient set of control edges and corresponding interference edges, as a form of proof-based learning. Blocking clauses derived from these edges are added back to the SAT formula to avoid enumeration of other path programs guaranteed to be correct, thereby improving performance and scalability. We have applied SMPP in the F-Soft program verification framework, to verify properties of real-world C programs that require path-sensitive reasoning. Our results indicate that the precision from analyzing individual path programs, combined with their efficient enumeration by SMPP, can prove properties as well as indicate potential violations in the large.

Categories and Subject Descriptors: D.2.4(Software/Program Verification):Assertion checkers, F.3.1(Specifying and Verifying and Reasoning about Programs):Assertions, F.3.2(Semantics of Programming Languages):Program analysis.
Terms: Languages, Verification

Keywords: Program Analysis, Abstract Interpretation, Path Programs, Symbolic Execution, SATisfiability Solvers, SMT solvers.

1. Introduction

Path-sensitivity is often a crucial requirement for verifying safety properties of programs. As it is infeasible to enumerate and analyze each path individually, analyses compromise by soundly merging information about executions along multiple paths. However, this results in a loss of precision, which may lead the analysis to determine falsely that a violation is possible. We present the *Satisfiability Modulo Path Programs* (SMPP) approach to program analysis for property verification. Our approach decomposes the verification of a given program to verification of its component *path programs* [4]. A path program represents a set of program executions, all of which traverse the same set of edges in a control flow graph, but may vary in the number of iterations of loops/recurrences encountered. Each path program in a given Control Flow Graph (CFG) is associated with a simple path in the MSCC (Maximal Strongly Connected Component) decomposition of the CFG, obtained by compacting loops and recurrences in the program into components [9]. Whereas the number of control paths in the original CFG may be infinite (due to loops and recurrences), the MSCC decomposition is acyclic with finitely may control paths. Nevertheless, this number can be prohibitively large for real-world programs. In our experiments, we have observed *millions* of path programs even for moderate-sized CFGs with about 500 control edges. Thus an explicit enumeration of path programs is not feasible.

We therefore propose *a SAT-based symbolic encoding* of the set of control paths associated with path programs. Starting from a SAT formula that represents all unexplored control paths in the MSCC graph that can reach an error location, we enumerate a control path and use an *oracle* to verify the corresponding path program. Each enumerated path program can be analyzed using "proof techniques" such as abstract interpretation [11, 12], or "falsification techniques" such as bounded model-checking that search for concrete error traces of violations [5]. We assume (w.l.o.g.) that a verification oracle presents Floyd-Hoare style proofs in the form of inductive invariants, or concrete witnesses upon violation.

We present a *proof-based learning* technique that avoids enumerating a large set of "related" path programs by reusing the proof of a single path program. The proof-based learning technique extracts a set of *sufficient edges* from a given path program and an associated set of *interference edges*. Our technique guarantees that any other path program that also traverses the same set of sufficient edges and none of the interference edges is also correct. As a result, our approach extends the *core reasons* for the proof of a given path program to apply to a large number of closely related path programs. The sufficient and interference edges are encoded as *blocking clauses* and added back to our SAT formula. We continue to enumerate solutions to this SAT formula until no more solutions can be found. Using proof-based learning significantly reduces the total number of path programs enumerated in SMPP, in the most dramatic case from *millions* to *hundreds*.

Our SAT encoding can be viewed as a control-flow abstraction with refinement performed by proof-based learning of sufficient/interference edges in the CFG. We do not encode or directly abstract data values. Reasoning over data is performed by the verification oracles. Note also that our enumeration operates over path programs and not program traces.

We present an instantiation of our approach that utilizes the abstract interpretation framework as the oracle of choice to prove properties over path programs. The inductive invariants obtained through abstract interpretation are used to extract sets of sufficient

POPL'10, January 17–23, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-479-9/10/01. . . $10.00

edges. We also present the use of over-approximate symbolic execution as another oracle that is efficient over path programs without loops or wherein the property to be proved is independent of the loops. It is possible to use other known techniques as oracles, such as lazy abstraction with interpolants [26] or predicate abstraction refinement [2, 3, 17].

We implemented the SMPP technique in the F-Soft program verification framework [21]. The implementation uses a symbolic execution engine based on Yices [16], and abstract interpretation engines using a succession of numerical domains — intervals [10], octagons [27], symbolic intervals [30] and polyhedra [13]. We evaluated our implementation by using it to verify array overflow and string library usage properties for C programs. We used publicly available benchmarks – smaller programs in Zitser et al. [34], and larger open source programs such as *openssh*, *thttpd* and *xvidcore*. Our evaluation shows that SMPP can derive proofs of properties that are beyond the reach of path-insensitive static analysis (already implemented in F-Soft), and it also identifies numerous potential violations. In comparison to the BLAST tool (using predicate abstraction with interpolant-based refinement [3]), SMPP can prove a majority of the properties proved by BLAST and identify additional violations in these programs, within a fraction of the time taken by BLAST.

Analogy with SMT Solvers: To prove safety properties, SMPP integrates program analysis oracles that prove properties about individual path programs with a SAT solver, which is used to enumerate over a Boolean abstraction of the control-flow of the program. This is similar in spirit to *Satisfiability Modulo Theory* (SMT) solvers [15, 16, 29]. To check the satisfiability of a given formula, an SMT solver integrates theory solvers (that check the satisfiability of conjunctive formulas over a theory) with a SAT solver that enumerates over a Boolean abstraction of the entire formula.

To solve a particular conjunctive formula, an SMT procedure employs theory solvers as oracles. Similarly, SMPP uses verification techniques over path programs as oracles. In fact, we may use different oracles for different path programs, based on observed characteristics (e.g. whether or not a backward slice w.r.t the property contains loops). Furthermore, the characteristics of the oracles affect the characteristics of our overall scheme (soundness, completeness, performance, etc.).

Theory-based learning in SMT solvers adds *conflict clauses* back to the SAT solver to refine the Boolean abstraction. Analogously, SMPP utilizes proof-based reasoning to derive sufficient edges and interference edges. These are added as blocking clauses to a SAT formula to block other enumerations by the SAT solver. By relying on a Boolean formula abstraction, we can apply SAT-based decision heuristics and blocking clauses that can drive the symbolic search over control paths in a property-driven manner. This is potentially advantageous compared to an *a priori* fixed search order on the CFG such as depth-first search (as in typical symbolic execution based approaches) or breadth-first search (as in typical BMC-based approaches).

The main contributions of this paper are: (a) SMPP: An approach for symbolic enumeration of path programs, using a SAT solver and proof-generating oracles for verifying path programs. (b) Proof-based learning techniques to identify and re-use a set of sufficient edges that serve as proofs of correctness. (c) A concrete instantiation of the SMPP approach, where we use abstract interpretation over different abstract domains and symbolic execution as oracles. (d) An experimental evaluation of the implementation, which shows that SMPP can prove more properties than path-insensitive static analysis and improve performance over predicate abstraction refinement. In particular, SMPP avoids the cost of an

Algorithm 1: Satisfiability Modulo Path Programs.

```
Input: $\Pi$ : Program CFG, $\Psi$ : $\langle n_f, \varphi \rangle$ assertion to be
       verified.
Result: Alarms : Set of possible path programs violating
        property.
begin
1    $\Pi_D$ := MSCCDecomposition($\Pi$)
2    $\Lambda_\Pi$ := SatEncodePaths($\Pi_D, n_0^D, n_f^D$)
3    while ( IsSatisfiable($\Lambda_\Pi$) ) do
4        $\pi$ := SolveAndObtainPathProgram($\Lambda_\Pi$)
5        $\eta$ := AnalyzePathProgram($\pi, \varphi$) /* Oracle */
6        if ($\eta(n) \models \varphi$) then      /* Proof obtained */
7            ($S_\pi, I_\pi$) := ExtractSufficientSet($\pi, \eta$)
         else                  /* violation obtained  */
8            ($S_\pi, I_\pi$) := PathSlice($\pi, n, \varphi$)
9            Alarms := Alarms $\cup$ {$S_\pi$}
10       $\Lambda_\Pi$ := $\Lambda_\Pi \wedge$ BlockingClause($S_\pi, I_\pi$)
11
end
```

expensive data abstraction-refinement process and divergences on loops, while fully utilizing advances in SAT/SMT solvers.

1.1 SMPP Approach At a Glance

Algorithm 1 presents the main steps of the SMPP algorithm. We now step through it with an example. Fig. 1 presents a simple imperative program fragment that computes a buffer length bLen based on an input pointer p, its length pLen, and an extra flag mode. The CFG representation is also given (also Cf. Example 2.1). The goal is to prove the unreachability of the CFG node labeled 15 corresponding to the program assertion at line 15. Fig. 2 depicts the major steps in the application of SMPP to the program, described as follows.

Extract an unexplored path program: SMPP first constructs the MSCC decomposition Π_D of the program and encodes the set of all paths in Π_D (Algo. 1 lines 1 – 2). It then chooses an unexplored path from this set (line 4), from which it constructs path program (A), illustrated as the path program highlighted in Fig. 2(A). Considering just the nodes and edges that lie on this path program, the unreachability of 15 can be proved by a *Constants*-domain analysis (Algo. 1 lines 5–6).

Extract a set of sufficient edges: The proof of unreachability of node 15 for path program (A) involves the *invariant* $p = 0$ that is valid at node 13 along the path program (A) (the invariant *does not*, in fact, hold for the program as a whole). SMPP applies a technique (detailed in Sec. 3.2) that pinpoints the key "reason" for the invariant. Such a "reason" takes the form of a set of *sufficient (control-flow) edges*. The set of sufficient edges (denoted S_1) corresponding to the proof in the first path program is shown in Fig. 2(B).

Any path program from 1 to 15 that traverses all of the edges in set S_1 will satisfy the property. Note that this includes paths that traverse the loops in the program arbitrarily many times as well. Computing sufficient sets therefore allows us to extend our proof along a single path program to multiple path programs. This step corresponds to Algo. 1 line 7.

Extract another path program: SMPP now seeks a path program that visits edges E such that $S_1 \not\subseteq E$. The search for such a path program is performed using a SAT solver (Algo. 1 lines 10, 3), and yields the second path program, as shown in Fig. 2(C). Now, the analysis must directly reason about the loop in this path program

```
0:  proc. foo (int * p, int pLen, int mode)
1:      int o, L := 1, bLen := 0;
2:      if (pLen < 1) return;
3:      if (p == NULL)
4:          pLen := -1;
5:      end-if
6:      if (mode)
7:          o := 1;
        else
8:          o := 0;
9:      end-if
10:     while( L ≤ pLen )
11:         if (o > 0)
12:             bLen := L - o;
13:         L := 2 * L;
14:     end-while
15:     ASSERT ( !p || bLen ≤ pLen);
```

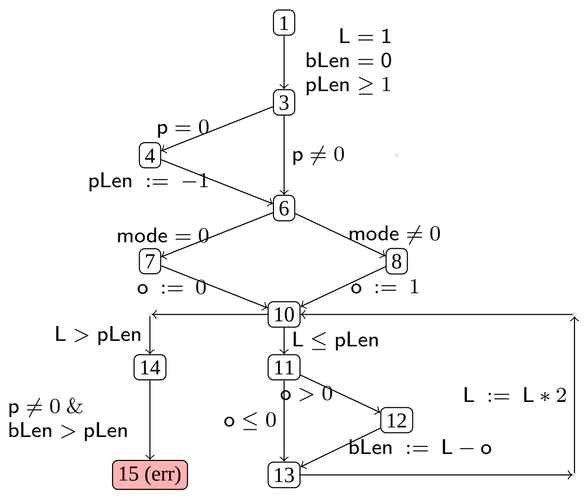


Figure 1. A simple imperative program and its CFG representation.

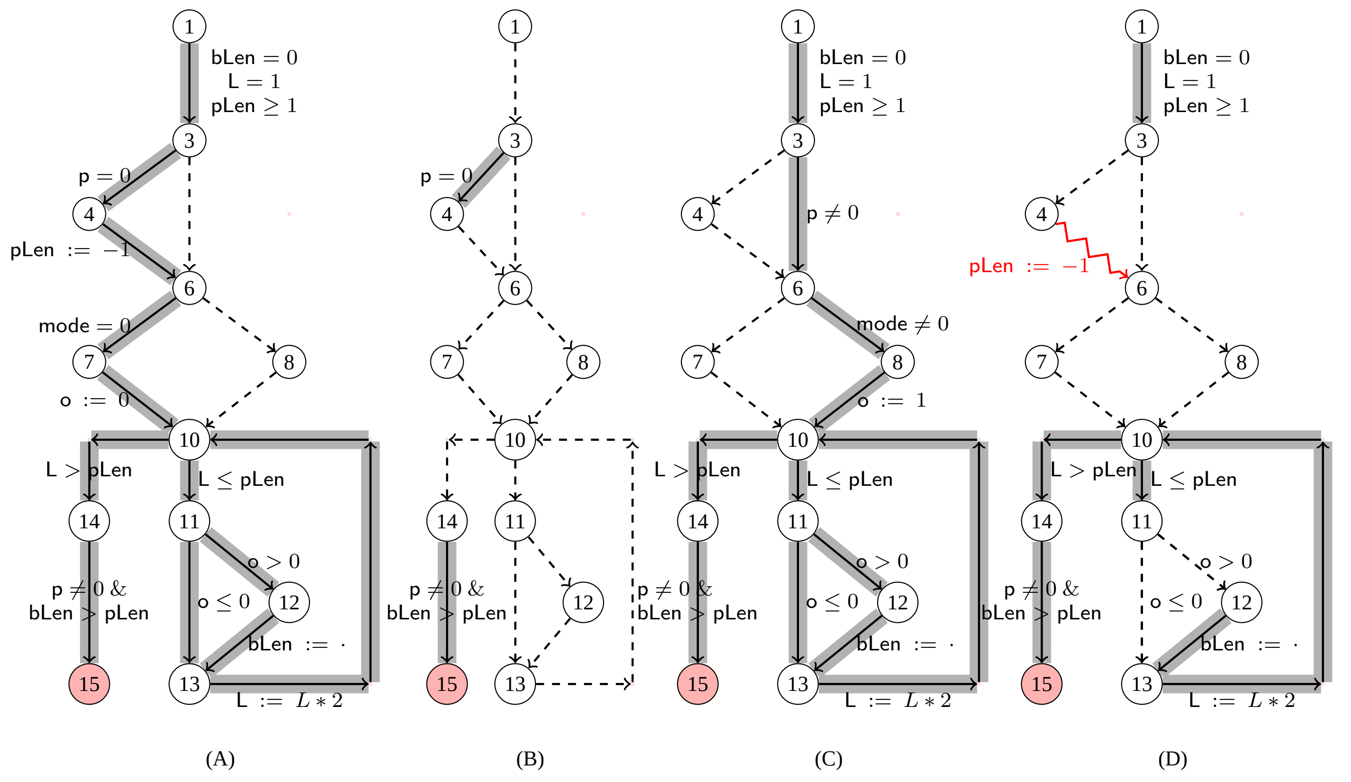


Figure 2. Path programs enumerated on running example by SMPP. **A:** Initial path program; **B:** Sufficient edges for (A); **C:** Second path program; and **D:** Sufficient (shaded) and interference (zigzag) edges for (C).

to prove unreachability of node 15. It can do so automatically by using the *octagon domain analyzer* [27] to establish the invariant bLen $\leq$ pLen at node 14 (Algo. 1 lines 5 – 6). This invariant proves correctness of this path program. The loop invariant bLen $\leq$ pLen computed at node 10 suffices to prove the property at node 15. Note that this invariant is independent of the variable o.

SMPP then extracts a set of sufficient edges S_2 for the invariant (detailed in Sec. 3.2), shown in Fig. 2(D). It also performs an interference analysis for S_2 over the entire program, to discover that the assignment to pLen on edge $4 \rightarrow 6$ may cause some paths that traverse all edges in S_2 not to preserve the property. This edge thus forms the *interference set* for sufficient set S_2, denoted as $I_2 = \{4 \rightarrow 6\}$ (Cf. Sec. 3.4). Any path program that traverses all the edges in S_2 but not the interference edge $4 \to 6$ satisfies the property.

Termination: SMPP has now computed sufficient-interference pairs of sets $(S_1, \emptyset)$ and (S_2, I_2) , extracted from the two path programs considered. *Every* path program is covered by these sets, i.e, each program path must either traverse every edge in S_1 , or traverse every edge in S_2 while traversing no edge in I_2. Recall that the edges that are part of the loop $10 \leadsto 10$ are part of an MSCC and thus not considered in enumerating path programs. As a result, by examining 2 out of the 4 possible path programs, SMPP has established the property. This computation occurs over Algo. 1 lines 10, 3, 11.

Symbolic Encoding: Fig. 2 consists of a small program with a few path programs. In practice, we have observed CFGs with as few as 500 edges that exhibit *millions* of path programs. Therefore, reasoning about sets of path progams explicitly is not feasible.

We overcome this difficulty by means of a symbolic encoding of control paths associated with path programs. The analysis uses a succinct propositional (Boolean) encoding that supports the following operations efficiently:

1. Encode all of the control paths between two nodes (in the MSCC graph) as a Boolean formula. Such an encoding is linear in the size of the CFG. Sec. 3.1 details this encoding.
2. Given a set P of unexplored control paths, represented as a Boolean formula Λ_P, and a sufficient-interference pair of edge sets (S, I), subtract all of the paths from P that traverse all the edges in S and none of the edges in I. Subtraction is performed by adding *blocking clauses* to the formula Λ_P.
3. Determine that there are no control paths left to examine, or produce an unexplored path. This is equivalent to checking the satisfiability of the updated Boolean SAT formula Λ_P.

SAT solvers have made impressive advances in the past decade, enabling them to support these operations for formulas with hundreds of thousands of variables and clauses. In this respect, our encoding is empirically shown to be quite amenable to existing solvers such as ZChaff [28], even for large programs.

2. Preliminaries

We first present our approach on single-procedure programs without function calls. We assume that all variables are of type *integer*. The handling of function calls (including recursion), non-integer types (especially pointers) and other features present in a programming language such as C is discussed in Sec. 4. Control-flow graphs (CFG) are used to represent imperative programs.

Control-Flow Graph A CFG $\Pi = \langle N, E, V, \rho, n_0, n_f, \Theta\rangle$ is a tuple where N is a set of nodes, $E \subseteq N \times N$ is a set of edges, $n_0 \in N$ is an initial location, $n_f \in N$ is a final location, and V is a set of program variables. Each edge $e \in E$ is labeled by a *transition relation* $\rho_e(V, V')$, a first-order assertion over *current-state* variables V and *next-state* variables denoted by V'. The first-order assertion Θ specifies the initial values of the program variables.

An execution of the CFG starts at the initial location with *initial* values to the program variables that satisfy Θ, and terminates when it reaches the final location. A *program assertion* $\langle n, \psi\rangle$ for $n \in N$ and assertion ψ over the program variables, requires that ψ hold whenever control reaches the node n.

Example 2.1. *Fig. 1 shows a CFG of a program over integer-valued and pointer variables (note that for this example, we are only interested in the integer value of the pointer itself). The nodes in the CFG are numbered according to the corresponding labels in the program. We wish to prove the unreachability of node* 15 *which corresponds to the* failure *of the assertion in the corresponding line of the program. However, a* path-insensitive *analysis is unable to prove the property. The join operation at line 5 may lose the relation between* p *and* pLen.

2.1 Path Programs

Given a graph $G : \langle N, E\rangle$, a *strongly connected component* (SCC) consists of a subset of nodes $C \subseteq N$ such that for each $n_1, n_2 \in C$, there exists a path from n_1 to n_2 and vice-versa. A strongly connected component C is *maximal*(MSCC) iff no strict superset of C is a strongly connected component. The MSCC-decomposition of a graph G is a directed, acyclic graph G_D whose nodes $N = \{n_1^d, \ldots, n_m^d\}$ correspond to each of the MSCCs C_1, $\ldots, C_m$ of G and whose edges connect n_i^d and n_j^d if and only if some node in C_i connects to some node in C_j by an edge in E. The MSCC decomposition of a graph can be computed in linear time [9]. The CFG in Fig. 1 contains a non-trivial MSCC consisting of the nodes $\{10, 11, 12, 13\}$.

Def. 2.1 (Path Program [4]). *Let Π_D be the MSCC decomposition of a CFG Π. A* path program $\pi : n_0^d \leadsto n^d$ *is a simple path in Π_D from the initial node n_0^d to a node n^d. The path program π naturally corresponds to a subset of nodes and edges in Π, obtained as the union of the MSCCs traversed by π and the edges that connect them.*

2.2 Inductive Invariants

Let Σ denote the universe of concrete program states and Γ be a domain of assertions over the program variables ordered by the logical implication $\models$. Each assertion $\varphi \in \Gamma$ represents a set of states $[\![\varphi]\!] \in 2^\Sigma$. An example of an assertion language used in practice is linear arithmetic formulae over the integer variables of a program.

Inductive Map For a set N of nodes in a control-flow graph, a *flow-sensitive* map $\eta : N \to \Gamma$ maps each node to a set of program states, represented as a formula. Such a map is *inductive* iff the following conditions hold:

$$\begin{array}{rcl}
\text{Initiation} & : & \Theta \models \eta(n_0) \\
\text{Consecution} & : & \text{foreach edge } e : \ n_i \to n_j, \\
 & & \eta(n_i) \wedge \rho_e(V, V') \ \models \ \eta(n_j)[V \mapsto V']
\end{array}$$

In order to establish a property $\langle n, \varphi\rangle$ for a given program, we seek an inductive map η such that $\eta(n) \models \varphi$.

2.3 Abstract Interpretation

Abstract Interpretation is a powerful and general framework for systematically computing inductive maps for a given program. An inductive map is a fixed point of a monotone operator in a suitable *abstract domain*. An abstract domain is a lattice $\langle \Gamma, \models\rangle$ that usually denotes an assertion language used to represent sets of states. The abstraction function $\alpha : 2^\Sigma \mapsto \Gamma$ and concretization function $\gamma : \Gamma \mapsto 2^\Sigma$ link elements of the abstract lattice to concrete sets of states. They are assumed to form a *Galois Connection* [12].

To compute an inductive map for the set N of control-flow nodes, with initial node n_0, we start from an initial map $\eta^0 : N \to \Gamma$ and iterate to produce a sequence of maps $\eta^1, \eta^2, \ldots$, wherein

$$\eta^0(n) : \left\{ \begin{array}{l} \top,\ n = n_0, \\ \bot,\ \mathsf{else} \end{array} \right. \quad \eta^{i+1}(n_b) : \left\{ \begin{array}{l} \top,\ n_b = n_0, \\ \bigsqcup_{n_a \to n_b} \mathsf{post}\ \begin{array}{l}(\eta^i(n_a), \\ n_a \to n_b)\end{array} \end{array} \right.$$

The iteration terminates when $\forall n \in N.\ \eta^{i+1}(n) \models \eta^i(n)$. Convergence is guaranteed if the domain L satisfies the *ascending chain* *condition*. Failing this, *widening* and *narrowing* operators can be used to guarantee convergence. If η is the fixed point obtained upon convergence, then $\gamma \circ \eta$ is an inductive invariant map.

Abstract domains such as the *interval domain* [10], *octagon domain* [27], *symbolic intervals* [30] and *polyhedra* [13] can be used to compute useful invariants over the program variables in order to prove properties of interest. Together, these domains represent various levels of trade-off between the strength of the invariants against the complexity of the analysis.

3. Path Program Enumeration

Let $\Pi = \langle N, E, V, \rho, n_0, n_f, \Theta\rangle$ be a control flow graph with MSCC decomposition $\Pi_D : \langle N_D, E_D\rangle$. Let $\Psi : \langle n, \varphi\rangle$ be a property under verification. We present a procedure for symbolically enumerating all control paths from n_0 to n_f, verifying Ψ along each associated path program.

Consider Algorithm 1, which shows the overall verification algorithm. The major steps of the algorithm are (a) encoding the set of (unexplored) control paths as a Boolean formula Λ_Π (line 2), (b) iterating over the solutions of Λ_Π (line 3), (c) analyzing each path program associated with the current solution (line 5) , (d) extracting sufficient sets, if correct (line 7), otherwise handling violations (line 8). The rest of this section presents the key steps in detail [1].

3.1 SAT Encoding

Path programs are represented by *simple paths* in the MSCC graph Π_D of the CFG Π. We now present a Boolean SAT-based encoding of all control paths in Π_D. This encoding corresponds to an implementation of $\mathsf{SatEncodePaths}$ and $\mathsf{SolveAndObtainPathPrograms}$ used in Algo. 1.

For each edge e in Π_D, the propositional variable p_e indicates if e belongs to the path program. Let $\mathsf{out}(m)$ denote the outgoing edges in E_D for a node $m \in N_D$ and $\mathsf{in}(m)$ denote the incoming edges. Let $\mathsf{src}(e)$ and $\mathsf{tgt}(e)$ denote the source and target nodes of an edge $e \in E_D$, respectively. For a set $E_s \subseteq E_D$, the formula $\mathsf{exactlyOne}(E_s)$ denotes that exactly one edge from E_s occurs in a path: $\mathsf{exactlyOne}(E_s) : \bigvee_{e\in E_s} p_e \wedge \bigwedge_{e,f\in E_s, e\neq f} p_e \Rightarrow \neg p_f$. The formula Λ_Π is the conjunction of all Boolean constraints given in Table 1. The constraints shown in the table enforce that any path program has (a) a visit to the initial node n_0, (b) a visit to the final node n_f, (c) exactly one incoming edge to each node n visited (with the exception of n_0), and (d) exactly one outgoing edge from a node n that is visited (with the exception of n_f).

Theorem 3.1. *The Boolean formula Λ_Π encodes all paths between n_0 and n_f in the MSCC decomposition Π_D of Π.*

3.2 Extracting Sufficient Sets

We now describe $\mathsf{AnalyzePathPrograms}$ and $\mathsf{ExtractSufficientSet}$ as used in Algo. 1. Let Π be the original CFG, n_0 be the initial node and $\langle n_f, \varphi\rangle$ be the property we wish to establish. Given a path program $\pi : n_0^D \rightsquigarrow n_f^D$, SMPP analyzes π using a proof-generating oracle. If the property holds over π, we extract a set of sufficient edges. Otherwise, failure to prove the property indicates a potential violation – details of handling violations are provided in Sec. 3.6. We now present the extraction of sufficient edges from proofs.

Let $\Pi_\pi : \langle N_\pi, E_\pi\rangle$ represent the subset of the original CFG induced by the path program π. The proof-generating oracle yields a proof of safety in the form of an inductive invariant map (fixed-point) $\eta : N_\pi \mapsto \Gamma$ over G_π. The map η maps each node $n \in N_\pi$ to an invariant $\eta(n)$, valid over all executions along π. If $\eta(n) \models \varphi$, then none of the executions leading from n_0 to n_f along the nodes and edges in π lead to a violation.

We extract sufficient edges which form the "core reason" behind our proof as follows: (A) we first prune away *unnecessary invariants* from the map η using the notion of a minimal supporting set. Removing such invariants is essential to obtain a compact set of sufficient edges. (B) We then compute an inductive map η' that supports this set. (C) From η', we directly extract a sufficient set of edges.

Example 3.1 (Unnecessary Invariants). *Consider the* "second" *path program π_2 shown in Fig. 2(C):*

$$1 \to 3 \to 6 \to 8 \to \{10, 11, 12, 13\}^* \to 14 \to 15\,.$$

A polyhedral domain abstract interpreter computes the following invariant:

$$\eta(14) : \left[\begin{array}{l} \mathsf{p} \neq 0 \wedge \mathsf{pLen} \geq 1 \wedge \\ 2\cdot\mathsf{bLen} \leq \mathsf{L} \wedge \underline{\mathsf{bLen} \leq \mathsf{pLen}} \wedge \\ \mathsf{mode} \neq 0 \wedge \mathsf{o} = 1 \end{array}\right].$$

This invariant establishes the required unreachability of node 15 *for the path program π_2. Nevertheless, the entire invariant is* not *required for the proof. The sole invariant* $\mathsf{bLen} \leq \mathsf{pLen}$ *(underlined above) suffices. The remaining invariants are extraneous.*

In principle, we can use other techniques that (attempt to) generate minimal sets of invariants required to prove a given property [6, 18, 19]. These techniques can generate strong invariants for small but complex loops. However, they are currently unsuitable for generating simple global invariants for large path programs. We provide a generic scheme described below utilizing a fixed point, wherein the fixed point can be computed in any way (including predicate abstraction over path programs).

Note. Removing unnecessary conjuncts just at the property node does not suffice. The removal of conjuncts from $\eta(13)$ in the example above now permits us to remove conjuncts from its predecessor node invariant $\eta(12)$, in turn spreading through the entire CFG.

Generalizing Invariants

To describe how to generalize an invariant map, we first describe the computation of *minimal support sets*, a key primitive. Let $Q : \{q_1, \ldots, q_m\}$ and q be assertions over program variables in a suitable logical theory, such that $q_1 \wedge q_2 \wedge \ldots \wedge q_m \models q$.

Def. 3.1 (Minimal Support Set). *A subset $Q' \subseteq Q$* supports *the inference $\bigwedge_{q_i\in Q} q_i \models q$ iff $\bigwedge_{q_j\in Q'} q_j \models q$. A* support set *$Q'$ is* minimal *iff no proper subset of Q' can support the inference.*

For $\psi_1 \models \psi_2$, let $\mathsf{MinSupport}(\psi_1, \psi_2)$ denote the set of minimal supporting conjuncts in ψ_1 that imply ψ_2. An implementation of $\mathsf{MinSupport}$ (through *unsatisfiable cores*) is available in existing solvers for many useful theories such as linear arithmetic.

Example 3.2. *The assertions $q_1 : i \geq j$, $q_2 : j \geq k+1$, $q_3 : i \geq k+1$, $q_4 : k \geq 1$ together imply the assertion $q : i \geq 2$ in the theory of linear arithmetic over integers. Note that the subset $\{q_1, q_2, q_4\}$ by itself (and no proper subset thereof) suffices to establish q and is thus a minimal support set. The minimal support set is not unique. The set $\{q_3, q_4\}$ also forms a minimal support set.*

We assume that the abstract domain Γ is a "Moore-closed domain." Specifically, each invariant φ is a finite conjunction of a set of atomic predicates that are negation closed: $\varphi : q_1 \wedge q_2 \cdots \wedge q_m$. Inductive invariants that consist of only conjunctive assertions suffice, in general, to prove any given property. Proofs involving disjunctions of conjunctions can be transformed into purely conjunctive proofs on a suitable *elaboration* of the original program [31].

[1] Proofs of key results are provided in the extended version of this paper available from the authors upon request.

Table 1. Boolean encoding of the set of all path programs (i.e, paths through the MSCC DAG).

Fact	Encoding
n_0 should be visited	$\mathsf{exactlyOne}(\mathsf{out}(n_0))$
n_f should be visited	$\mathsf{exactlyOne}(\mathsf{in}(n_f))$
Source of each edge has exactly one predecessor (except for n_0)	$\bigwedge_{e\in E_D,\ \mathsf{src}(e)\neq n_0}\ [p_e \Rightarrow \mathsf{exactlyOne}(\mathsf{in}(\mathsf{src}(e)))]$
Target of edge has exactly one successor (except for n_f)	$\bigwedge_{e\in E_D,\ \mathsf{tgt}(e)\neq n_f}\ [p_e \Rightarrow \mathsf{exactlyOne}(\mathsf{out}(\mathsf{tgt}(e)))]$

Let η be a fixed-point map that establishes a property $\langle n_f, \varphi\rangle$, i.e., $\eta(n_f) \models \varphi$. Since η is a fixed-point, for every edge $e : n_1 \rightarrow n_2 \in E_\pi$, the following consecution condition holds:

$$\eta(n_1)\ \wedge\ \rho_e(V, V') \models \eta(n_2)[V \mapsto V'] \,.$$

Our overall strategy to generalize η is to construct a finite sequence of maps $\mu^0, \ldots, \mu^N$, wherein the initial map is defined as:

$$\mu^0(m) = \begin{cases} \mathsf{MinSupport}(\eta(n_f), \varphi) & m = n_f \\ \mathit{true} & m \neq n_f \end{cases} .$$

The initial map $\mu^0 : N_\pi \mapsto L$ maps the node n_f to the minimal support set that enables $\eta(n_f)$ to prove the required property $\langle n_f, \varphi\rangle$ and maps all other nodes to true. The iterative process $\mu^0 \ldots, \mu^N$ will converge onto a final map μ^N that establishes the property $\langle n_f, \varphi\rangle$, contains no redundant invariants, and generalizes η.

The intermediate maps μ^i for $i < N$ need not be inductive. For instance, the map μ^0 could fail the consecution property for the incoming edges to the node n_f. The maps μ^i, $i \in [0, N]$ have the following properties:

(a) μ^N is inductive and proves the property $\langle n_f, \varphi\rangle$.

(b) For each μ^i, and for each node m, the assertion $\mu^i(m)$ consists of a subset of conjuncts from $\eta(m)$. As a result $\eta(m) \models \mu^i(m)$.

(c) Each successive map incorporates at least as many conjuncts from η as the previous, i.e, $\forall\ i < j, m \in N_\pi.\ \mu^j(m) \models \mu^i(m)$.

We propose a process called *local repair* to derive the map μ^{i+1} from μ^i.

Local Repair: We address the failure of μ^i for $i < N$ to be an inductive invariant by means of *local repair*. To perform the local repair of μ^i, we strengthen $\mu^i(a)$ for some node a to address the failure of consecution along an edge $e : a \rightarrow b$: $\mu^i(a) \wedge \rho_{a\rightarrow b}(V, V') \not\models \mu^i(b)$. However, $\eta(a) \wedge \rho_e(V, V') \models \mu^i(b)$ [2]. Let Q_a be a minimal subset of conjuncts from $\eta(a)$ that supports this implication. The *local repair* of $\mu^i(a)$ w.r.t $a \rightarrow b$ is $\mu^{i+1}(a) = \mathsf{MinSupport}(\eta(a), \mathsf{pre}(\mu^i(b), e :\ a \rightarrow b)$. The application of $\mathsf{MinSupport}$ removes redundant conjuncts from the assertion $\eta^i(a)$. Thus $\mu^{i+1}(a)$ minimally supports consecution across the edge $a \rightarrow b$. Strengthening $\mu^i(a)$ for some node a may invalidate the consecution condition for some of its incoming edges. A new repair iteration is then required to address this failure. This process converges when μ^i is inductive, thus needing no further iteration.

Theorem 3.2. *The process of repeated local repair terminates in finitely many steps yielding a fixed-point map μ, s.t. $\eta(b) \models \mu(b)$ for all $b \in N_\pi$.*

Example 3.3. *Consider again the path π from Ex.3.1. Abstract interpretation computes an invariant relating program variables, including* pLen, bLen, p, *and* L. *The result of the repair iteration for the invariant* $\mathsf{bLen} \leq \mathsf{pLen}$ *at node* 14 *leads to the map* μ *partially depicted as:*

n	3	10	11	13&14
$\mu(n)$	$\mathsf{bLen} = 0$ $\mathsf{pLen} \geq 1$ $\mathsf{L} = 1$	$\mathsf{bLen} \leq \mathsf{pLen}$	$\mathsf{L} \leq \mathsf{pLen}$ $\mathsf{bLen} \leq \mathsf{pLen}$	$\mathsf{bLen} \leq \mathsf{pLen}$

The result μ of the repair iteration is used to extract a set of *sufficient edges,* $S_\pi \subseteq E_\pi$ that are sufficient for the proof of Ψ.

Def. 3.2 (Sufficient Edges). *An assignment* $e : a \xrightarrow{x\,:=\,\mathsf{e}} b$ *is a* supporting *edge w.r.t* μ *if* $\mu(b)$ *contains an invariant assertion involving the variable* x [3]*. A condition* $e : a \xrightarrow{q(\mathsf{e})} b$ *is a supporting edge if* $\mu(a) \not\models q(\mathsf{e})$ *and* $\mu(b) \models q(\mathsf{e})$.

A *sufficient set* for μ is the set of all such edges. The definition of sufficient edges immediately implies a method of deriving such a set of edges from a map μ.

Example 3.4. *Continuing Ex.3.3, the sufficient edges corresponding to the proof over* π *consist of the assignments* $1 \rightarrow 3$, $12 \rightarrow 13$ *and* $13 \rightarrow 10$ *along with conditions* $10 \rightarrow 11$, $10 \rightarrow 14$ *and* $14 \rightarrow 15$.

3.3 Over-Approximate Symbolic Execution

In many cases, the path program may not contain loops, or the loops present do not affect the property of interest. In such cases, we propose to use symbolic execution along the path program as an oracle. The power of symbolic execution lies in the ability of fast SMT solvers to reason about the feasibility of large formulae in theories such as linear arithmetic or bit-vectors, and in case of infeasibility to quickly extract minimal unsatisfiable cores. Note that standard symbolic execution in general cannot reason about all paths through a program loop. Therefore, we use an *over-approximate symbolic execution* (described below). If it succeeds in proving the property, we directly obtain a sufficient set. If it fails, then we resort to a more general proof technique like abstract interpretation.

An over-approximate symbolic execution of the path program π constructs a formula ψ_π in a suitable logical theory. This formula is derived by composing the transition relations along the edges in the path program π. Assignments belonging to loops are treated as assigning a non-deterministic value, and conditions present in loops are treated as nondeterministic choices. Finally, for the property $\langle n, \varphi\rangle$, we assert $\neg\varphi$ as a condition encountered at the node n.

Theorem 3.3. *If the over-approximate symbolic execution of a path program π yields an unsatisfiable formula ψ_π, then any execution of the path program π satisfies $\langle n, \varphi\rangle$.*

Example 3.5. *Symbolic execution of the "first" path program* $\pi_1 : 1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow \{10, 11, 12, 13\}^* \rightarrow 14 \rightarrow 15$, *shown in Fig. 2(A) (originally from Ex.2.1) yields the following formula*

[2] $\because \eta(a) \wedge \rho_e(V, V') \models \eta(b) \models \mu^i(b)$

[3] Alternatively, the consecution $\mathsf{post}(\mu(a), e) \models \mu(b)$ should *cease* to hold if the assignment is made non-deterministic.

obtained by composing the transition relations of the individual edges:

$$\left[\begin{array}{l} \rho_{1,3} : (\mathsf{bLen}_0 = 0 \wedge \mathsf{pLen}_1 \geq 1 \wedge \mathsf{pLen}_1 \geq \mathsf{pLen}_0) \wedge \\ \rho_{3,4} : (\underline{\mathsf{p}_0 = 0}) \wedge \rho_{4,6} : (\mathsf{pLen}_1 = -1) \wedge \\ \rho_{6,7} : (\mathsf{mode}_0 = 0) \wedge \rho_{7,10} : (\mathsf{i}_0 = 0) \wedge \\ \rho_{10,14} : (\mathsf{L} > \mathsf{pLen}_1) \wedge \\ \rho_{14,15} : (\underline{\mathsf{p}_0 \neq 0} \wedge \mathsf{bLen}_1 > \mathsf{pLen}_1) \end{array}\right]$$

The subscripts on the variables occurring in the transitions are derived from an SSA-form of the program ξ or using a use-def chain analysis. Note that ψ_ξ is infeasible, proving that the path ξ satisfies Ψ.

Let $\psi : \rho_1 \wedge \cdots \wedge \rho_m$ be an infeasible formula obtained from a path program π. Furthermore, let $R = \{\rho_{i_1}, \ldots, \rho_{i_k}\}$ be an *unsatisfiable core* for the formula ψ and $S = \{e_{i_1}, \ldots, e_{i_k}\}$ be the subset of edges that yield the transitions in the set R. The set S forms a sufficient set for the infeasibility of π.

Example 3.6. *Returning to Ex.3.5, the unsatisfiable core consists of the transition relation $\rho_{3,4}$ along with $\rho_{14,15}$. This yields the sufficient set $S = \{3 \rightarrow 4, 14 \rightarrow 15\}$.*

3.4 Interference Analysis

Thus far, we have focused on the analysis of a single path program. Interference analysis extends the learning due to sufficient edges extracted from a given path program, to consider other path programs. Thus, interference analysis operates on the entire CFG.

For a path program π, the sufficient set $S_\pi \subseteq E_\pi$ represents edges relevant to the proof of the property along π. Using S_π, we seek to characterize the set of path programs in *the original CFG* that are also guaranteed to satisfy the property Ψ and are proven by the same sufficient set. To do so, we also need to reason about potentially *interfering assignments* in the CFG. This corresponds to the component of $\mathsf{ExtractSupportSet}$ from Algo. 1 that yields I_π.

Example 3.7. *In Ex.3.4, the process of local repair over the path program π_2 in Fig. 2(C) yields a sufficient set*

$$S_2 : \{1 \rightarrow 3, 10 \rightarrow 14, 10 \rightarrow 11, 12 \rightarrow 13, 13 \rightarrow 10, 14 \rightarrow 15\}\,.$$

The path π_1 shown in Fig. 2(A) traverses all of the edges in the set S_2, yet the proof of the property obtained along the sufficient set S_2 does not apply for this path. The reason is that the assignment $\mathsf{pLen} := -1$ *along the edge $4 \rightarrow 6$ invalidates the value of the variable* pLen *that is initially defined in $1 \rightarrow 3$. Therefore the proof in Ex.3.4 does not apply to this path.*

This assignment disrupts the critical use-def *chain between edges $1 \rightarrow 3$ where the variable* pLen *is defined and $10 \rightarrow 11$, where it is used. Our approach is to identify a set of interference edges that can invalidate the use-def chains in the sufficient set.*

Def. 3.3 (Interfering Edge). *An assignment edge $e \notin E_\pi$ assigning variable x is an* interference edge *for a sufficient set $S_\pi \subseteq E_\pi$ iff there exist edges $e_1, e_2 \in S_\pi$ wherein e_1 defines x, e_2 uses x, and a path of the form $e_1 \rightsquigarrow e \rightsquigarrow e_2$ exists in the original CFG.*

Returning to Ex.3.7, we verify that the edge $4 \rightarrow 6$ interferes with the def-use chain $e_1 : 1 \rightarrow 3$ and the condition $e_2 : 10 \rightarrow 11$. Given a path program π and the sufficient edges S_π, a set of interfering edges I_π can be computed using a use-def chain computation and a control reachability analysis on the original CFG. In principle, a finer semantic criterion for interference can be formulated that checks whether an interfering edge preserves the invariants related to the sufficient set.

However, in our implementation, the syntactic criterion presented here is used due to its simplicity.

Theorem 3.4. *If a property $\Psi : \langle n, \varphi \rangle$ holds on a path program π then it holds on any path program $\xi : n_0 \rightsquigarrow n$ visiting* all the edges *in S_π and* none of the edges *in I_π.*

3.5 Blocking Clauses

We apply the oracles described in Secs. 3.3 and 3.2, in sequence, to a path program π in an attempt to obtain a sufficient set S_π. If the oracle obtains a proof, then the interference analysis described in Sec. 3.4 yields a sufficient-interference pair (S_π, I_π) consisting of the sufficient edges and their corresponding interference edges. We now describe $\mathsf{BlockingClause}$ from Algo. 1 that applies these sets to rule out other paths with the same proofs of correctness.

In general, propositions in our Boolean encoding describe path program edges. However, our sufficient or interference edge sets may contain edges inside loops (as in Ex.3.7). Following our interpretation of path programs, a path program that visits the representative node n of an MSCC also traverses all the edges inside the MSCC. Therefore, for the sake of convenience, we introduce a propositional formula $\varphi_e :\ p_e \equiv \bigvee_{e' : n' \rightarrow n \in E_D} p_{e'}$ for every edge e occurring inside an MSCC represented by node n. Note that p_e can be used as a proposition that models a visit to the representative node n with the addition of a Boolean formula φ_e above, relating p_e to the other propositions in our encoding.

The formula $\mathsf{BlockPaths}(S_\pi, I_\pi) :\ \bigvee_{e \in S_\pi} \neg p_e\ \vee\ \bigvee_{f \in I_\pi} p_f$, encodes paths that either (a) do not visit every node of S_π, or (b) visit some node of I_π. Adding this formula as a *blocking clause* to Λ_Π avoids revisiting the same set of sufficient edges. This provides proof-based learning to avoid the enumeration of related path programs.

3.6 Handling Violations

We now describe how to handle violations as depicted in Algo. 1 lines 8 – 9. One could halt the enumeration upon encountering a violation, but it may be desirable to continue the search for errors that stem from a different cause. To this end, SMPP obtains a *path slice* for a violation along the lines of Jhala and Majumdar [24]. As a result of path slicing, the analysis obtains a set $S_\pi \subseteq E_\pi$ of edges that cause the error a set I_π of edges that may interfere with the use-def chains in S_π. Our treatment of the pair (S_π, I_π) obtained from a potential violation is identical to that obtained from a proof through repair iteration.

Theorem 3.5 (Jhala and Majumdar [24]). *Let S_π, I_π be as computed by the path slicing technique. For any path ξ that visits all the nodes in S_π and none of the nodes in I_π, the path ξ violates the property $\langle n, \varphi \rangle$ iff π does.*

3.7 Expanding Loops

A path program, corresponding to a control path in the MSCC decomposition, treats loops monolithically. A given loop is entirely part of a path program wherein arbitrarily many iterations are considered, or alternatively no iterations are considered. In the case when the entire program consists of a single while loop (as is the case with control programs), our technique is equivalent to running the oracle over the entire program. This imprecision can be remedied by unwinding and unrolling a given loop, so that our techniques can reason about specific paths in the loop. Furthermore, our choice of an MSCC decomposition was intended to create an acyclic graph related to the CFG. Our approach can be generalized to use other schemes for creating an acyclic graph such that the paths in this graph are related to fragments in the original CFG.

4. Implementation

We have implemented our approach as a part of the F-Soft program verification platform for C programs [21, 22]. F-Soft checks C programs for buffer overflows, string API usage, NULL pointer dereferences, user-defined type-state properties, memory leaks and so on. For a detailed description of our modeling of structures, pointers and arrays, see [22].

F-Soft Framework

The F-Soft front-end *flattens* structure and union types into simple types, constructs a memory model by providing magic number addresses to storage locations, and instruments the program for properties being checked. Pointer aliasing and arithmetic are handled by modeling their effects over integer variables based on a flow-insensitive points-to analysis.

Example 4.1. *Fig. 3 illustrates the construction of a memory model in F-Soft based on the results of a points-to analysis. Our model replaces each local variable p in a function f by a variable $f : p$ with global scope. If f can be called in a recursive context then such a variable is treated as a* summary variable.

Corresponding to each pointer variable p, we introduce an instrumentation variable star(p) to track the contents of its store. The value of star(p) is non-deterministic if p does not point to a valid location. The expression $*p$ is replaced by its representative star(p). An assignment to the L-value *$*p$ is rewritten into an assignment to all the variables x that p can potentially point to. Such an assignment to x is guarded by a condition $p == \&x$ that enforces the points-to relationship. Further details and rationale are provided elsewhere [21, 22].*

The initial CFG of the program is simplified considerably through program slicing and constant folding. We then perform a series of flow and context sensitive analyses such as constant folding, interval analysis, and various numerical domain analyses. F-Soft implements many abstract domains in a partially path-sensitive abstract interpretation framework [1, 30]. The analysis uses these domains in combination or in succession to attempt to prove a property, each run of the analysis reusing the invariants obtained by the previous runs. F-soft re-slices the program model based on the properties proved by static analysis, reducing it further. At the end of static analysis, the final CFG with its unproven properties is provided as an input to the SMPP implementation.

SMPP Implementation

Our implementation follows the description in Algorithm 1 with modifications to accommodate function calls. The treatment of function calls and returns ensures that the enumerated path programs properly match calls and returns; and different paths inside a function can be traversed by a path program upon multiple visits under different calling contexts. This is achieved by *context numbering* the function calls in the CFG. The context numbering scheme described by Whaley and Lam is used [32]. Each CFG edge e then yields multiple propositions $p(e, c)$ based on the different calling contexts c that the edge may be visited in. The encoding for incoming edges at call sites and outgoing edges at returns ensure that enumerated calling block context with a return to the appropriate call site. The rest of the encoding remains unchanged

Recursive calls are currently unwound up to a specified depth and then replaced by a call to a function that returns a non-deterministic value and has non-deterministic side-effects on variables passed by reference. In practice, this seems to have a minimal impact on the checking of runtime errors.

The enumerated path programs are first analyzed using over-approximate symbolic execution and then (if needed) by the abstract interpreter and a local repair iteration. Currently, the support set and unsatisfiable-core computation are performed using the SMT solver Yices [16]. A failure to prove a property is currently reported as a *potential violation*. In the future, we plan to integrate our technique with a model checker or a *concolic execution* engine over path programs to concretize these potential violations.

Table 2. Legend for abbreviations used in Table 3.

Abbrv.	Remark
Blk	Number of blocks at start of SMPP.
Prp	Number of properties (asserts).
Prf	Number of Proofs.
PP tot	Number of path programs total (estimated).
PP enum	Number of path programs enumerated by SMPP.
PP Proofs	Number of path programs proved correct.
SE	Symbolic Execution
AI	Abstract Interpretation
RI	Repair Iteration

5. Experimental Evaluation

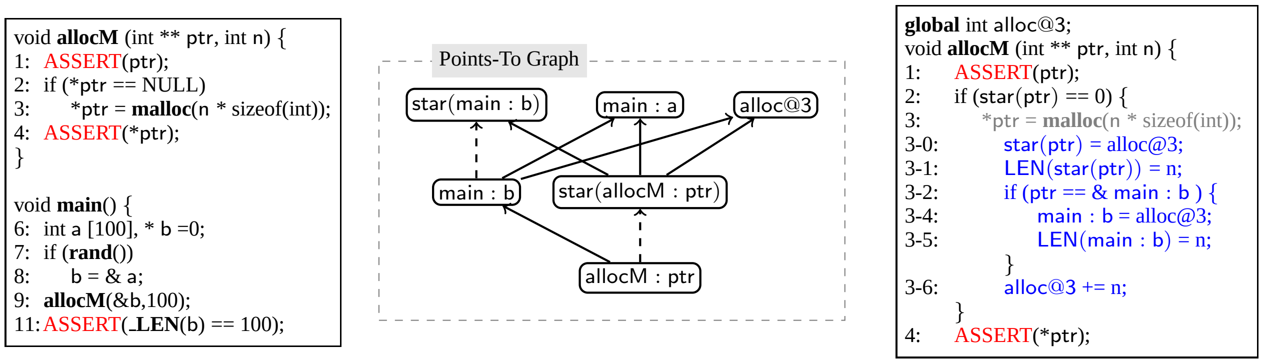
We conducted experiments to address the following: (a) how efficiently SMPP can verify common safety properties that require path-sensitive reasoning, (b) the effectiveness of proof-based learning for proving many program-paths safe with the analysis of a few, and (c) the power and efficiency of SMPP against another path-sensitive analysis, predicate abstraction.

Fig. 4 presents our experimental setup using the F-Soft framework. The experiments consist of processing the given C program through our front-end which includes path-insensitive, flow- and context-sensitive abstract interpretation through numerical domains such as *constants*, *intervals*, and *octagons*. Properties proved using these analyses are *removed* from the CFG, followed by re-slicing and simplification. Thus all properties on which SMPP is tested require some degree of path-sensitive reasoning to validate.

Table 3 presents our experimental results on the Zitser et al. benchmarks [34]. These programs consist of important buffer overflow bugs found in commonly used programs such as `wu-ftpd`, `bind`, `nslookup` and so on, along with the fixes made to them. Note, however, that F-Soft automatically instruments many more properties, as compared to the single line of code that is marked in these benchmarks. The abbreviations used in Table 3 are expanded in Table 2. Table 3 reports the size of each program at the start of SMPP, number of properties, the number of proofs (per oracle), and the time taken (total, percentage for stages of the analysis). Each property that is not a proof is reported as a potential witness along with a sliced path program. For some these potential witnesses, the symbolic execution reported a path program slice that did not include a loop, indicating a concrete witness. The results demonstrate that SMPP can prove a significant number of path-sensitive properties and identify potential witnesses efficiently. Note that symbolic execution is successful in proving a majority of the paths efficiently, and that abstract interpretation is key for a significant portion of properties that involve loops.

In Table 3, the Col. Path Programs (Tot) presents an estimate of the total number of path programs, estimated using SAT (in many cases, the estimator timed out after an hour) while Path Programs (Enum) presents the number actually enumerated by SMPP. The difference demonstrates the effectiveness of proof-based learning.

Table 4 compares the performance of SMPP against Blast v2.5 [3]. To focus on the analyses (rather than on differences in program modeling), we printed the CFG on which SMPP is applied as a C program, using goto statements to enforce control flow. The variables in this program are all integers (generated after the F-Soft



```
void allocM (int ** ptr, int n) {
1:  ASSERT(ptr);
2:  if (*ptr == NULL)
3:      *ptr = malloc(n * sizeof(int));
4:  ASSERT(*ptr);
}

void main() {
6:  int a [100], * b =0;
7:  if (rand())
8:      b = & a;
9:  allocM(&b,100);
11: ASSERT(_LEN(b) == 100);
```

```
global int alloc@3;
void allocM (int ** ptr, int n) {
1:      ASSERT(ptr);
2:      if (star(ptr) == 0) {
3:          *ptr = malloc(n * sizeof(int));
3-0:        star(ptr) = alloc@3;
3-1:        LEN(star(ptr)) = n;
3-2:        if (ptr == & main : b ) {
3-4:            main : b = alloc@3;
3-5:            LEN(main : b) = n;
            }
3-6:        alloc@3 += n;
        }
4:      ASSERT(*ptr);
```

Figure 3. A simplified illustration of F-Soft's memory modeling for a program with pointers.

Table 3. Experimental Evaluation on the Zitser et al. Benchmarks on CFG obtained after initial static analysis. The path counter timeout was set to 1hr. Table 2 explains the abbreviations used here.

Name	LOC Orig.	Blk (tot)	Prp (tot)	Prf	Path-Programs Total	Path-Programs Enum	PP Proofs Tot	PP Proofs SE %	PP Proofs AI %	Times (seconds) Tot sec	Times (seconds) SE %	Times (seconds) AI %	Times (seconds) RI %
SENDMAIL													
s2-ok	1151	744	60	20	>1.2M	204	131	80	20	136.76	7	65	21
s2-bad	1132	721	58	21	>1.2M	259	195	86	14	139.17	10	77	3
s4-ok	776	263	12	6	12	12	6	0	100	10.98	2	95	1
s4-bad	713	284	30	15	30	30	15	0	100	30.29	1	96	0
s5-ok	837	179	14	13	14	2	1	100	0	0.5	40	0	0
s5-bad	810	182	17	16	17	2	1	100	0	0.5	40	0	0
s6-ok	317	121	8	3	8	8	3	0	100	13.18	0	98	0
s6-bad	315	121	8	3	8	8	3	0	100	14.91	0	98	0
s7-ok	1824	1357	174	59	>.5M	415	224	92	8	132.4	25	48	0
s7-bad	1816	1347	170	94	>.6M	412	271	92	8	135	24	53	0
BIND													
b1-ok	2177	670	55	10	>.4M	214	168	96	4	179.6	9	85	0
b1-bad	2117	662	54	9	>.4M	181	135	95	5	171.79	9	86	0
b2-ok	2706	957	80	11	>.4M	285	216	97	3	282.23	23	69	0
b2-bad	2688	955	80	11	>.4M	289	220	97	3	230.14	33	59	0
WU-FTPD													
f1-ok	628	144	13	4	188	28	14	71	29	11.81	2	88	8
f1-bad	562	178	21	9	481	71	52	92	8	25.14	3	91	4
f2-ok	1208	119	9	5	5637	17	13	100	0	0.25	68	20	0
f2-bad	936	114	8	4	2175	14	10	100	0	0.19	72	16	0

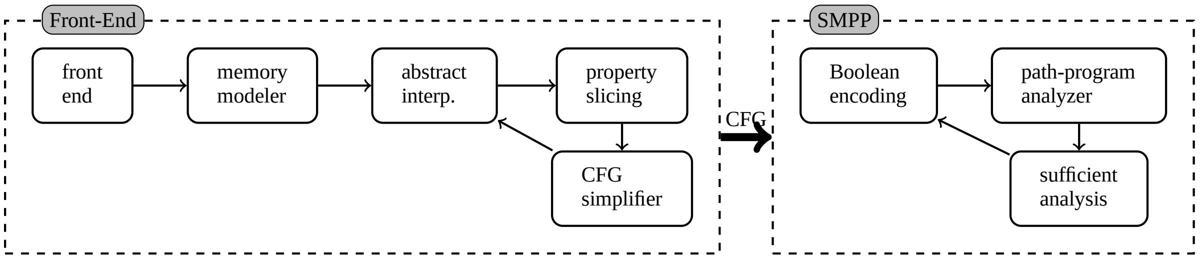


Figure 4. Experimental setup: Front-end includes path-insensitive analyses, followed by SMPP.

front-end process of instrumentation, memory modeling, simplifications, slicing and static property proofs). Blast had to be invoked multiple times on the same program, each instance targeting a different property. This was needed to avoid the default aggregation of properties in Blast which failed with errors in many instances. For fair comparison, we ran SMPP multiple times with each instance targeting a single property, recomputing the SAT encodings and the sufficient sets from scratch. The comparison in Table 4 shows that while Blast can prove more properties than SMPP in some cases, our coarser control-based abstraction can be much faster and finds more violations in many cases. A majority of the failures by Blast resulted from an explosion in the number of predicates on

Table 4. Comparison of SMPP with BLAST. Prp: number of properties (asserts), Wit: number of concrete witnesses, and Fail: Blast failures. Note that SMPP is run multiple times (once for each property) for fairer comparison.

Name	Prp	SMPP		BLAST			
		Prf	Time	Prf	Wit	Fail	Time
s2-ok	60	20	2m10s	35	0	25	2h 7 m
s2-bad	58	21	2m25s	35	0	23	1h50m
s4-ok	12	6	0m11s	11	1	0	0h1.3m
s4-bad	30	15	0m31s	17	1	12	2h 8 m
s5-ok	14	13	0m00.5s	6	0	8	0h55m
s5-bad	17	16	0m00.5s	9	0	8	1h32m
s6-ok	8	3	0m14s	4	0	4	0h17m
s6-bad	8	3	0m15s	4	0	4	0h17m
s7-ok	174	59	28m	135	1	38	17h22m
s7-bad	170	94	27m	134	1	35	15h55m

loops (termed "Gremlins"). Our abstract interpretation followed by a proof-based learning seems to be adequate for many such cases.

Analysis of Larger Benchmarks. Table 5 summarizes our experimental results on larger case studies. The experimental setup remains the same as in Fig. 4. Program sources were downloaded from the internet and first analyzed using our tool SpecTackle [22], which infers likely preconditions and post-conditions corresponding to pointer and array accesses in the functions [4]. F-Soft (with SMPP) is then invoked for each function in the program. The preconditions for the entry functions are assumed, whereas the preconditions for called functions are asserted. In order to control the CFG size, calls to functions not reachable from the entry function within a context of depth 4 were replaced by non-deterministic choice. Table 5 shows the results. The SMPP approach is invoked only in those cases where there are unresolved properties from the initial path-insensitive analyses. This accounts for roughly 40% of the functions, on average. The total time taken by the SMPP approach is of the same order as that taken by the initial model construction and static analysis phases. In almost all cases, the overall analysis (initial+SMPP) terminates within the given time limit of 45 minutes. Note that SMPP detects a significant number of extra proofs, over and above sophisticated flow- and context-sensitive polyhedral abstract interpretation techniques. We did not run Blast on these larger case studies, as it would have required significant manual effort in instrumenting the CFGs generated by our front-end.

6. Related Work

Abstraction Refinement . Our technique fits broadly into the abstraction refinement paradigm. The Boolean formulae representing all unexplored control paths is a coarse control-flow abstraction. This abstraction is successively refined by eliminating the path programs proved correct.

The main differences from typical abstraction refinement approaches are: (a) Our abstraction is based solely on control locations. This is a very inexpensive abstraction to compute, deferring the heavier work to an oracle for checking correctness of a path program corresponding to the enumerated control path. Other approaches typically use an explicit representation of the control flow with data predicates, e.g. Boolean programs [2, 3]. Such abstractions are more expensive to compute than our abstractions. (b) Rather than a refinement loop over false error traces (counterexamples) [2, 8], our refinement loop operates over path programs associated with the enumerated control paths. (c) Our approach avoids *divergences* on loops in the program. This is because we enumerate over an acyclic MSCC-based graph derived from the CFG, wherein each control path corresponds to a path program that can potentially capture infinitely many control paths on the original CFG. It is well known that effective handling of loops is a stumbling block for many abstraction refinement techniques. Abstract interpretation techniques using widening have been well-optimized to prove common types of run-time properties, even in the presence of loops. (d) The decomposition of the overall program into multiple path programs allows flexibility in choosing a suitable oracle for individual sub-problems. In our implementation, we use symbolic execution to handle path programs without loops, and abstract interpretation to handle path programs with loops. Other oracles, such as predicate abstraction refinement, can also be used.

Our work is closely related to recent work by Heizmann et al [20] . They propose an abstraction refinement scheme for *trace abstractions*. An over-approximation of the set of possible traces is successively refined by means of an *interpolant automaton* that recognizes a set of infeasible traces. The interpolant automata are also derived from proof techniques that can generate Floyd-Hoare style inductive invariants. The main differences include: (a) our technique operates on path programs as opposed to error traces; (b) we employ a symbolic encoding using SAT to keep track of unexplored path programs, as opposed to an explicit representation.

Lazy abstraction with interpolants by McMillan [26] provides a lazy scheme to refine an abstract model on demand, by utilizing interpolants derived from refuting paths in the program. This work also avoids the cost of an expensive abstraction. However, the refinement is driven by error traces, the control flow is handled explicitly, and details of proof-based learning are different from our technique.

The notions of path programs in our work are directly inspired by the work of Beyer et al. [4]. That work also employs path programs and invariants to avoid loop divergences. In contrast, (a) we use *Boolean formulae* abstracting just the control flow, as opposed to *predicate abstraction* (Boolean programs), and (b) we refine through *blocking clauses* rather than using *invariants as predicates*. Using blocking clauses for refinement is potentially more scalable than using invariants as predicates. First, invariant generation techniques typically generate a large set of invariants, a majority of which are redundant. Beyer et al. [4] do not attempt to minimize the set of invariants. Our work provides this reduction by means of the *local repair iteration* discussed in Section 3. Second, each predicate added can potentially double the complexity of model checking an abstraction, whereas our conflict clauses are small and seem to have very little impact on the size of the Boolean formula abstraction. Another approach is to use abstract interpretation to derive useful program invariants as a pre-processing step, to avoid expensive refinement iterations over loops [23].

Bounded model checking of programs. Efficient SAT-based techniques have been used for bounded model checking (BMC) [5] of programs in tools such as CBMC [7], F-Soft [21], and for scalable summary-based analysis in Saturn [33]. These techniques automatically utilize SAT-based conflict analysis and learning for pruning the search space. However, they suffer in the presence of loops, which require deep unwindings that result in large SAT problems. Furthermore, BMC typically handles all paths up to some bounded length as a single monolithic problem. In contrast, we encode only the control paths as a SAT formula, which is much smaller than a typical BMC formula that encodes unwindings of a program.

Abstract interpretation and path-sensitive analysis. Other approaches to path sensitive analysis include *ESP* [14], *trace partitioning* [25], *elaborations* [31], amongst many others. These techniques employ heuristics to control the trade-off between perform-

[4] These benchmarks along with the inferred preconditions are available upon request.

Table 5. Results on Larger Open-Source Case-Studies

Name	KLOC	#Fun	#Prec	Front-End				SMPP					
				Blk	Time	Prp	Prf	#Fun		Blk	#PP	Prf.	Time
				avg	sec	tot	tot	tot	timeout	avg	tot	tot	tot, sec
thttpd-2.25b	14.7	172	1901	263	5162	12161	11069	68	1	199	1622	512	1393
ssh-server-4.1	30.1	313	2047	190	7755	42773	41182	127	5	120	1681	564	1197
xvidcore	63.9	350	6127	520	13259	17219	12020	190	6	331	5220	2090	15728

ing a join operation or a logical disjunction at the merge points in the CFG. However, the join vs. disjunction choice is *inferred* in our SMPP scheme by the disjunction-based decomposition over control paths.

7. Conclusion

We have presented the Satisfiability Modulo Path Programs (SMPP) approach to program analysis, which lifts to the architecture of SMT solvers to path-sensitive program analysis. We have demonstrated it to be effective in analyzing real-world programs.

References

[1] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivancic, Ou Wei, and Aarti Gupta. SLR: Path-sensitive analysis through infeasible-path detection and syntactic language refinement. In *SAS*, volume 5079 of *LNCS*, pages 238–254, 2008.

[2] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In *CAV*, volume 2102 of *LNCS*, pages 260–264, 2001.

[3] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.

[4] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309. ACM, 2007.

[5] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207, 1999.

[6] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20(4-5):379–405, 2008.

[7] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, 2004.

[8] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00*, pages 154–169. Springer, 2000.

[9] Thoman H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.

[11] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to Abstract interpretation, invited paper. In *PLILP '92*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252, 1977.

[13] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *POPL'78*, pages 84–97, January 1978.

[14] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[15] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[16] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[17] Sussane Graf and Hasan Saidi. Construction of abstract state graphs with PVS. In *CAV'97*, volume 1254 of *LNCS*, pages 72–83, 1997.

[18] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[19] Ashutosh Gupta and Andrey Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640, 2009.

[20] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *Static Analysis Symposium (SAS'09)*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.

[21] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *CAV*, volume 3576 of *LNCS*, pages 301–306. Springer, 2005.

[22] Franjo Ivančić, Sriram Sankaranarayanan, Ilya Shlyakhter, and Aarti Gupta. Buffer overflow analysis using environment refinement. Draft (2009), Available Upon Request.

[23] Himanshu Jain, Franjo Ivančić, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV'06*, pages 137–151, 2006.

[24] Ranjit Jhala and Rupak Majumdar. Path slicing. In *PLDI*, pages 38–47, 2005.

[25] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05*, volume 3444 of *LNCS*, pages 5–20, 2005.

[26] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, volume 4144 of *LNCS*, pages 123–136, 2006.

[27] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053 of *LNCS*, pages 155–172. Springer, May 2001.

[28] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[29] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(*T*). *J. ACM*, 53(6):937–977, 2006.

[30] Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Program analysis using symbolic ranges. In *SAS*, volume 4634 of *LNCS*, pages 366–383, 2007.

[31] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, volume 4134 of *LNCS*, pages 3–17. Springer, 2006.

[32] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer analysis using binary decision diagrams. In *PLDI'04*. ACM Press, June 2004.

[33] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[34] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. SIGSoft/FSE'04*, pages 97–106. ACM, 2004.