Structural Invariants*

Ranjit Jhala[1] Rupak Majumdar[2] Ru-Gang Xu[2]

[1]UC San Diego [2]UC Los Angeles

Abstract. We present *structural invariants* (SI), a new technique for incrementally overapproximating the verification condition of a program in static single assignment form by making a linear pass over the dominator tree of the program. The 1-level SI at a program location is the conjunction of all dominating program statements viewed as constraints. For any k, we define a k-level SI by *recursively strengthening* the dominating join points of the 1-level SI with the $(k-1)$-level SI of the predecessors of the join point, thereby providing a tunable selector to add path-sensitivity incrementally. By ignoring program paths, the size of the SI and correspondingly the time to discharge the validity query remains small, allowing the technique to scale to large programs. We show experimentally that even with $k \leq 2$, for a set of open-source programs totaling 570K lines and properties for which specialized analyses have been previously devised, our method provides an automatic and scalable algorithm with a low false positive rate.

1 Introduction

An invariant at a program location is a (first-order) predicate over the program state that holds whenever the location is visited during execution. Thus to prove that a programmer-specified assertion always holds at a location, it suffices to check if any invariant implies the asserted predicate. Verification-conditions (VC) are a powerful technique for generating invariants, and hence verifying properties of programs [18, 13, 16]. However, the use of VCs has been hindered by several considerations. First, in order to generate the VC, the fixpoint semantics of every loop in the program must be provided as *loop invariants*. Second, in order to be precise, VC generators encode all execution paths of the program. When applied to large programs, this results in large formulas that cannot be solved efficiently. Thus, while *generic*, in that they are applicable to any user specified assertion, and *precise*, in that they capture all path correlations, the use of VC-based techniques has been limited to proving deep properties of programs, often with substantial manual intervention. For checking properties over large code bases, researchers typically develop specialized analyses based on dataflow analysis or abstract interpretation, which use a fixpoint computation to find the semantics of the program over a fixed abstraction. These techniques often sacrifice genericity and precision to gain automation and scalability: they use

* This research was sponsored in part by the NSF grants CCF-0427202 and CNS-0541606.

property-specific abstractions to gain automation and thus are not generic; they gain scalability by merging execution paths at join points, leading to imprecision in the form of false alarms.

In this paper, we consider a middle ground. We present a lightweight VC generation technique that is automatic and scalable enough to prove many useful safety properties over large code bases, without requiring an expert to devise a specialized analysis for each program and each property, and yet is precise enough to capture many structural idioms used by the programmer to ensure correctness, even in the presence of path correlations typically missed by dataflow tools. We achieve this using *structural invariants* (SI), a series of increasingly precise *over-approximations* of the VC, which can be efficiently computed from the *dominator tree* of the program's control-flow graph (CFG) in *static single assignment* (SSA) form. SIs use the dominator tree to capture control flow information and the SSA form to capture data flow information about the program. By using these well-optimized compiler techniques, and by incrementally refining approximate VCs, our algorithm scales to large code bases. By not requiring explicitly provided loop invariants but using simple approximations, our algorithm is automatic. While this restricts the properties we can prove, we provide empirical evidence that shows extremely coarse approximations suffice to prove a large variety of useful properties on many large applications. In particular, we show for a set of different safety properties considered in the software verification literature [19, 21, 20, 7, 14], our technique is generic, yet completely automatic and scalable, running in time comparable to specialized dataflow analyses, often with better precision.

The first and coarsest over-approximation (the 1-structural invariant) is obtained as the *conjunction* of the dominating operations' predicates. That this forms an invariant follows from two observations. First, the operations dominating the target location are guaranteed to execute on any path to the target. Second, if the program is in SSA form, then the variables occurring in a dominating operation will not be modified after the last occurrence of that operation on a path to the target. The 1-SI ignores the predecessors of control flow join points dominating the target. Hence, correlated conditional control flow to the target location is not tracked. To regain path-sensitivity that distinguishes between the executions prior to the join point, we *recursively strengthen* the predicate of the join using the *disjunction* of SIs of predecessors of the join. The degree of distinguishing or "branch-sensitivity" is parameterized: for any $k > 1$, the k-SI is obtained by strengthening the join points using the $(k-1)$-SI of the predecessors. By only strengthening join points (and not loop heads), we compute an SI by traversing a subset of the dominator tree in a single pass. For each $k > 0$, the k-SI provides an over-approximation of the VC, becoming more precise with increasing k. In the limit, *i.e.,* when k equals the number of CFG nodes, the SI is equivalent to the standard VC [16, 17] obtained by unrolling each loop of the program once and arbitrarily updating the loop-modified variables. The parameter k provides a tunable selector for statements that most influence the assertion. For example, the 1-SI includes only the operations that must happen on *all* CFG paths to the target, and the 2-SI captures one level of branching (required to prove, e.g., conditional locking behavior). Empirically, we have found the 1-SI to be two orders of magnitude smaller than a full VC that sweeps over the entire program, and the resulting validity queries are discharged up to two orders of magnitude faster than the queries for the full VC. Despite dropping the other constraints, we found that the 1-SI is sufficient to prove 70% of the assertions we examined, and for most of the remaining assertions, 2-SI sufficed.

To demonstrate the precision and genericity of our technique, we have implemented a tool `psi` that generates k-SIs, and used this to successfully analyze a diverse set of open-source programs for three important safety properties with a low false positive rate. `psi` takes as input a C program annotated with assertions, and a number k, and computes the k-SI for the program at each assertion point, and then uses SIMPLIFY [12] to discharge the validity query, and thus prove the assertion. The first property (studied in [20] using language-level techniques) checks the consistent use of *tag fields* when using unions inside structures in C programs. In the example of Figure 1(a), which is representative of networking code, the header field `h` corresponds to a TCP packet if the `proto` field has value `TCP` and is a UDP packet otherwise, and the property checks that at each cast to `TCP *` (resp. `UDP *`), `proto==TCP` (resp. `proto==UDP`). The second property (studied in [19, 24, 14]) checks that Linux drivers acquire and release locks in strict alternation. In most cases, each call to `unlock` is dominated by a call to `lock` and vice versa. As seen in Figure 3(a), in the few cases where branch sensitivity is required to capture some idiomatic uses like conditional locks and trylocks, the 2-SI suffices. The third property is for *privilege levels* (studied in [7]): at any point where a `suid` program calls `execv`, the effective user-id is non-root. In our experiments, system calls setting the user-id dominate the call to `execv` so the 1-SI suffices to prove these assertions. We have used `psi` to check these properties on a total of 570K lines of code containing 759 assertions. With $k \leq 2$, we proved 667 of these, and found 16 bugs and 76 false alarms. The total running time of all experiments was less than one hour. In contrast, the software model checker Blast took at least an order of magnitude more time on all experiments, and did not finish on several runs. We believe this demonstrates that lightweight VC-based techniques can be made as automatic and scalable as a variety of specialized analyses. While our coarse approximations may generate an invariant that is not strong enough to prove the property of interest, our experience is that SIs can be used as an effective pre-pass for any verification effort to "filter out" many assertions, leaving sophisticated program verification tools to focus their resources on more complicated properties.

2 Structural VC Generation

We formalize structural invariants for an imperative language with integer variables. We begin with the intraprocedural case.

Operations. Our programs are built using: (1) *assignment* operations `x := e`, which correspond to assigning the value of expression `e` to the variable `x`. A

```
struct iphdr {
  int check;
  int proto; // TCP or UDP
  ...
  char *h; // payload
};
example1(struct iphdr *ip) {
1:if (!ip->check) {
    ... // perform checks
2:  ip->check = 1; }
  ...
3:t = ip->proto;
4:if (t==TCP) {
5:  TCP *tcphdr = (TCP *)ip->h;
    ...
6:} else {
7:  UDP *udphdr = (UDP *)ip->h;
    ... }
  }
```

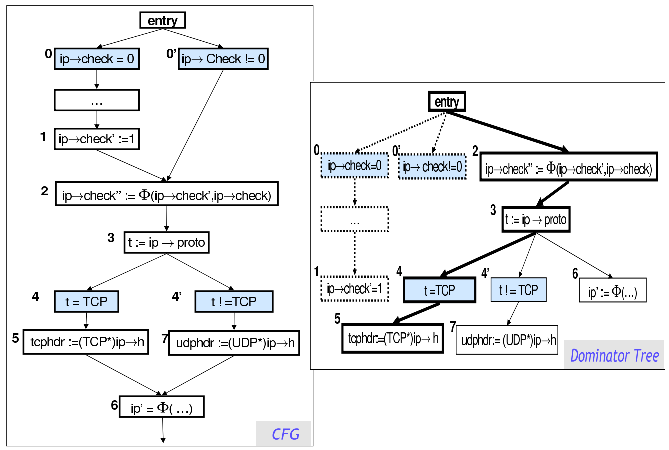

Fig. 1. (a) Example 1 (b) CFG in SSA form (c) Dominator Tree of CFG

basic block is a sequence of assignments. (2) *Assume* operations assume (p), which continue program execution if the boolean expression p evaluates to true, and halt the program otherwise.

Control-Flow Graphs. The control flow of a procedure is given by a *Control-flow Graph* (CFG), a rooted, directed graph $G = \langle N, E, \mathtt{n}_e, \mathtt{n}_x \rangle$ with:

1. A set of control nodes N, each labeled by a basic block or assume operation;
2. Two distinguished nodes: an *entry* node $\mathtt{n}_e$ and an *exit* node $\mathtt{n}_x$;
3. A set of edges $E \subseteq N \times N$ connecting control nodes: $(\mathtt{n}_1, \mathtt{n}_2) \in E$ if control can transfer from the end of $\mathtt{n}_1$ to the beginning of $\mathtt{n}_2$. We assume that $\mathtt{n}_e$ has no incoming edges, and $\mathtt{n}_x$ has no outgoing edges.

Let $\mathsf{pred}(\mathtt{n})$ denote the set $\{\mathtt{n}' \mid (\mathtt{n}', \mathtt{n}) \in E\}$ of *predecessors* of n in the CFG. We assume that the set $\mathsf{pred}(\mathtt{n})$ is ordered, and refer to the k-th predecessor of a node n to denote the k-th element in the ordering in $\mathsf{pred}(\mathtt{n})$. We write $\mathsf{vars}(\mathtt{n})$ to denote the set of variables appearing in the operation op labeling n. A *path* π of length m to a node n in the CFG is a sequence $\mathtt{n}_1 \ldots \mathtt{n}_m$ where $\mathtt{n}_1 = \mathtt{n}_e$, $\mathtt{n}_m = \mathtt{n}$, and for each $1 \leq i < m$ the pair $(\mathtt{n}_i, \mathtt{n}_{i+1}) \in E$. We denote by $\pi(i)$ the ith node $\mathtt{n}_i$ along the path. We denote by $\pi[j]$ the prefix of the path, $\mathtt{n}_1 \ldots \mathtt{n}_j$. A node n is *reachable* in the CFG if there is a path π to n. We assume that all nodes in N are reachable from $\mathtt{n}_e$.

Dominators. For two CFG nodes $\mathtt{n}, \mathtt{n}'$ we say n *dominates* $\mathtt{n}'$ if for every path π to $\mathtt{n}'$ of length m, there is some $1 \leq i \leq m$ such that $\pi(i) = \mathtt{n}$. We say n *strictly dominates* $\mathtt{n}'$, written $\mathtt{n} \mathrel{\mathsf{D}} \mathtt{n}'$, if n dominates $\mathtt{n}'$ and $\mathtt{n}, \mathtt{n}'$ are distinct. We write $\mathsf{D}(\mathtt{n})$ for the set $\{\mathtt{n}' \mid \mathtt{n}' \mathrel{\mathsf{D}} \mathtt{n}\}$. We write $\mathsf{D}^{-1}(\mathtt{n})$ for the set $\{\mathtt{n}' \mid \mathtt{n} \mathrel{\mathsf{D}} \mathtt{n}'\}$. We say n is the *immediate dominator* of $\mathtt{n}'$ if for every $\mathtt{n}'' \in \mathsf{D}(\mathtt{n}')$, we have $\mathtt{n}''$ dominates n. Each node n of the CFG has a unique immediate dominator which we write as $\mathsf{Idom}(\mathtt{n})$. A *dominator tree* is a rooted tree whose nodes are the nodes of the CFG, whose root is the entry node $\mathtt{n}_e$, and where the parent of a node $\mathtt{n}$ is $\mathsf{Idom}(\mathtt{n})$.

SSA. We assume that programs are represented in *static single assignment* (SSA) form [10], in which each variable in the program is syntactically assigned exactly once. Programs in SSA form have special ϕ-assignment operations of the form $\mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ that capture the effect of control flow joins. A ϕ-assignment $\mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ for variables $\mathtt{x}$, $\mathtt{x}_1, \ldots, \mathtt{x}_n$ at a node $\mathtt{n}$ implies: (1) $\mathtt{n}$ has exactly n predecessors in the CFG, (2) if control arrives at $\mathtt{n}$ from its jth predecessor, then $\mathtt{x}$ has the value $\mathtt{x}_j$ at the beginning of $\mathtt{n}$. Further, we distinguish two kinds of ϕ-assignments: those at the header of natural loops (denoted ϕ^ℓ), and the others (denoted ϕ).

Semantics. For a set of variables X, an X-state is a valuation for the variables X. The set of all X-states is written as $\mathsf{V}.X$. Each operation op gives rise to a transition relation $\overset{\mathsf{op}}{\rightsquigarrow} \subseteq \mathsf{V}.X \times \mathsf{V}.X$ as follows. We say $s \overset{\mathsf{op}}{\rightsquigarrow} s'$ if either $\mathsf{op} \equiv \mathtt{assume}\ (\mathtt{p})$, $s \models \mathtt{p}$, and $s' = s$, or $\mathsf{op} \equiv \mathtt{x} := \mathtt{e}$ and $s' = s[\mathtt{x} \mapsto s.\mathtt{e}]$. The relation $\overset{\mathsf{op}}{\rightsquigarrow}$ is extended to basic blocks by sequential composition. We say that a state s *can execute* the operation op if there exists some s' such that $s \overset{\mathsf{op}}{\rightsquigarrow} s'$. A formula φ over the variables X represents all X-states where the valuations of the variables satisfy φ. For a formula φ, we write $\mathsf{vars}(\varphi)$ for the set of variables appearing syntactically in φ. We say that φ' is a *postcondition* of φ w.r.t. an operation op if $\{s' \mid \exists s \in \varphi. s \overset{\mathsf{op}}{\rightsquigarrow} s'\} \subseteq \varphi'$, *i.e.,* executing op from a state satisfying φ results in a state satisfying φ'. We say that *a path* π *satisfies the formula* φ if φ is a postcondition of *true* w.r.t. the sequence of operations along π. For a CFG node $\mathtt{n}$ we say that a formula φ is an $\mathtt{n}$-*invariant* if *every* path π to $\mathtt{n}$ satisfies φ.

Operation Predicates. For an operation op, define an *operation predicate* $[\![\mathsf{op}]\!]$:

op	$[\![\mathsf{op}]\!]$	op	$[\![op]\!]$	op	$[\![\mathsf{op}]\!]$
$\mathtt{x} := \mathtt{e}$	$\mathtt{x} = \mathtt{e}$	$\mathtt{assume}\ (\mathtt{p})$	$\mathtt{p}$	$\mathsf{op}_1;\ \ldots;\ \mathsf{op}_n$	$\bigwedge_{i=1}^n [\![\mathsf{op}_i]\!]$
$\mathtt{x} := \phi(\mathtt{x}_1, , \ldots, \mathtt{x}_n)$	$\bigvee_{i=1}^n \mathtt{x} = \mathtt{x}_i$	$\mathtt{x} := \phi^\ell(\mathtt{x}_1, , \ldots, \mathtt{x}_n)$	*true*		

For a node $\mathtt{n}$ labeled with operation op, we write $[\![\mathtt{n}]\!]$ for $[\![\mathsf{op}]\!]$. For a program in SSA form, the operation predicate $[\![\mathsf{op}]\!]$ is a postcondition of *true* w.r.t. the operation op. Additionally, for a node $\mathtt{n}$ we define $\Phi(\mathtt{n}, j)$ to be $\mathtt{x} = \mathtt{x}_j$ if $\mathtt{n}$ is labeled $\mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_j, \ldots, \mathtt{x}_n)$ and $[\![\mathtt{n}]\!]$ otherwise. In other words, $\Phi(\mathtt{n}, j)$ constrains the variable assigned at a ϕ-node to the value held at the j-th predecessor of $\mathtt{n}$.

2.1 Structural Invariants

Dominator Invariants. We first relate dominator nodes in the CFG to program invariants. This provides an efficient algorithm to compute invariants. Our technique follows from three observations about dominators and programs in SSA form. First, immediately after an operation op is executed, the new state satisfies the operation predicate $[\![\mathsf{op}]\!]$. Second, if $\mathtt{n}'$ dominates $\mathtt{n}$, then along every execution path to $\mathtt{n}$, there is an instant, just after the dominator $\mathtt{n}'$ is executed, at which $[\![\mathtt{n}']\!]$ is satisfied. Third, if $\mathtt{n}' \mathsf{D} \mathtt{n}$, then in any execution path, after the *last* occurrence of $\mathtt{n}'$, the only nodes visited are those that are dominated by $\mathtt{n}'$ (this is illustrated in Figure 2(a)), and none of the variables in $\mathsf{vars}(\mathtt{n}')$ are ever modified. Thus, as $[\![\mathtt{n}']\!]$ held immediately after (the last occurrence of) $\mathtt{n}'$, it is preserved until execution reached $\mathtt{n}$. Hence $[\![\mathtt{n}']\!]$ is a $\mathtt{n}$-invariant. It follows that the conjunction of node predicates for all nodes dominating $\mathtt{n}$ is an $\mathtt{n}$-invariant. We call this the *dominator invariant* of $\mathtt{n}$.

Theorem 1. [**Dominator Invariants**] *For a node* $\mathtt{n}$ *of a CFG in SSA form, the formula* $[\![\mathtt{n}]\!] \wedge \bigwedge_{\mathtt{n}' \in \mathsf{D}(\mathtt{n})} [\![\mathtt{n}']\!]$ *is an* $\mathtt{n}$*-invariant.*

Example 1. [**Tagged-Union Verification**] Figure 1(a) shows an example of a C program that deserializes a stream of bytes to extract a packet. The packet is represented by the C structure `iphdr`, with a tag field `int proto` which specifies if the *payload* field `char *h`, a stream of characters, corresponds to a TCP or a UDP payload. Precisely, if `proto` is `TCP`, then `h` is a TCP payload, else `h` is a UDP payload. Figure 1(b) shows the CFG of the program in SSA form. For simplicity, we treat pointer accesses such as $\mathtt{ip}{\rightarrow}\mathtt{check}$ as unaliased scalars, our implementation handles pointers correctly. Since union types are not tagged explicitly in C, programmers use a tag field to determine the type of the union instance and then cast the data appropriately before access. However, absent or incorrect checks lead to data access bugs which are a common cause of hard to find bugs or crashes. This *data access specification* introduces implicit assertions in the code wherever the field `h` is accessed. For example, in Figure 1(a), there are two implicit assertions: one at line 5 where `h` is cast to a `TCP` pointer which asserts: $\mathtt{ip} \rightarrow \mathtt{proto} = \mathtt{TCP}$ and one at line 7 where `h` is cast to `UDP` pointer which asserts: $\mathtt{ip} \rightarrow \mathtt{proto} \neq \mathtt{TCP}$. To check correct usage of tagged unions, we must find a program invariant at these assertion points that implies the assertions. ∎

Example 2. The CFG in SSA form and the dominator tree for the example of Figure 1(a) are shown in Figures 1(b), 1(c). In Figure 1(c), we see that the nodes dominating $\mathtt{n}_5$ in Figure 1(b) are $\mathtt{n}_2, \mathtt{n}_3, \mathtt{n}_4$. By conjoining their respective operation predicates we get the dominator invariant:

$$(\mathtt{ip} \rightarrow \mathtt{check}'' = \mathtt{ip} \rightarrow \mathtt{check} \vee \mathtt{ip} \rightarrow \mathtt{check}'' = \mathtt{ip} \rightarrow \mathtt{check}') \wedge\ \mathtt{t} = \mathtt{ip} \rightarrow \mathtt{proto} \wedge \mathtt{t} = \mathtt{TCP}$$

which implies, and thus proves, the implicit data access assertion $\mathtt{ip} \rightarrow \mathtt{proto} = \mathtt{TCP}$ at 5. By virtue of the program being in SSA form, the dominator invariant captures the flow of value through the local variable `t`. ∎

ϕ-Strengthening. Dominator invariants ignore conditional control flow merges in the code and, as Example 3 below shows, are often not precise enough to prove properties of interest.

Example 3. In the networking example of Figure 1, suppose that we additionally wish to verify that the payload `h` is only accessed after the checksum has been verified (i.e., `check` field is set to a non-zero value). This yields the additional (implicit) assertions at statements `5:` and `7:` that $\mathtt{ip} \rightarrow \mathtt{check}'' \neq 0$. The conjunction of the operation predicates of the dominators of $\mathtt{n}_5$, namely $\mathtt{n}_2, \mathtt{n}_3, \mathtt{n}_4$ is insufficient due to the ϕ-node, $\mathtt{n}_2$ where control joins after the branch. At such a node, a variable may get a value from one of several predecessors, neither of which dominates the target node. So, as dominator invariants only conjoin operation predicates for dominating operations, they do not capture branch correlations. ∎

To gain path sensitivity, we recursively compute the invariant of each predecessor of a ϕ-node $\mathtt{n}$ (a join point) and take their disjunction to strengthn the node predicate of $\mathtt{n}$. While computing the invariant for the ith predecessor, we additionally conjoin the predicate $\Phi(\mathtt{n}, i)$, thus updating the value of each variable assigned at $\mathtt{n}$ to the value in the ith predecessor. We call this process *recursive ϕ-strengthening*. We explicitly parameterize the recursive ϕ-strengthening with a bound k. For $k = 1$ we get exactly the dominator invariants (there is no recursive strengthening), while for higher values of k we recursively strengthen using the $(k-1)$-SI of the predecessors of the ϕ-nodes.

Formally, we define k-structural invariants using two recursively defined functions Ψ and Γ. The function Ψ is defined for nodes $\mathtt{n}_r, \mathtt{n}$ and integer k as:

$$\Psi((\mathtt{n}_r, \mathtt{n}), k) \equiv [\![\mathtt{n}]\!] \wedge \bigwedge_{\mathtt{n}' \in \mathsf{D}(\mathtt{n}) \cap \mathsf{D}^{-1}(\mathtt{n}_r)} [\![\mathtt{n}']\!] \wedge \Gamma(\mathtt{n}', k)$$

if $k > 0$ and $\mathtt{n}_r \neq \mathtt{n}$ and *true* otherwise. Intuitively, the parameter $\mathtt{n}_r$ is the ancestor in the dominator tree whose subtree is being used to generate the SI for $\mathtt{n}$, and the parameter k is an explicit bound on the recursion depth. The function Γ is used for the recursive ϕ-strengthening. For node $\mathtt{n}'$ and integer k, if $k > 0$, and $\mathsf{pred}(\mathtt{n}') \cap \mathsf{D}^{-1}(\mathtt{n}') = \emptyset$, *i.e.,* $\mathtt{n}'$ is a join node (and not a loop header otherwise one of the predecessors would be dominated by $\mathtt{n}'$) then:

$$\Gamma(\mathtt{n}', k) \equiv \bigvee_{\mathtt{n}_j \in \mathsf{pred}(\mathtt{n}')} (\Phi(\mathtt{n}', j) \wedge \Psi((\mathsf{Idom}(\mathtt{n}'), \mathtt{n}_j), k-1)$$

and it is defined as *true* otherwise, *i.e.,* no strengthening is done. Recall that for a join ϕ-node, the formula $\Phi(\mathtt{n}', j)$ simply constrains the value of the "merged" variable to be that of the variable at the j-th predecessor of $\mathtt{n}'$. The *k-structural invariant* of a node $\mathtt{n}$ of the CFG is $\Psi((\mathtt{n}_e, \mathtt{n}), k)$. The *structural invariant* of a node $\mathtt{n}$ is $\Psi((\mathtt{n}_e, \mathtt{n}), N)$.

A k-SI "unfolds" the *nesting structure* of the program. The parameter k allows us to incrementally tune the precision of the invariant, and use coarser (and faster computed) invariants wherever possible. By raising k, we are increasing the *branch-width* sensitivity of the analysis, and setting k to the number of CFG nodes gives us the exact SI. This provides a dual approximation to the usual "bounded-depth" analyses, where all paths of length less than a certain bound are analyzed.

We use induction on k to prove that k-SI are invariants, Theorem 1 provides the base case.

Theorem 2. *[Structural Invariants] For every CFG $G = (N, E, \mathtt{n}_e, \mathtt{n}_x)$ in SSA form, $\mathtt{n} \in N$, and $k \in \mathbb{N}$, (1) the k-Structural Invariant of $\mathtt{n}$ is an $\mathtt{n}$-invariant, and (2) $\Psi((\mathtt{n}_e, \mathtt{n}), k+1) \Rightarrow \Psi((\mathtt{n}_e, \mathtt{n}), k)$.*

Figure 2 shows how the recursive strengthening works vis-a-vis the dominator tree and the CFG. The 1-SI conjoins the node predicates for each node in the path from the root node to the target node (shaded) in the dominator tree,

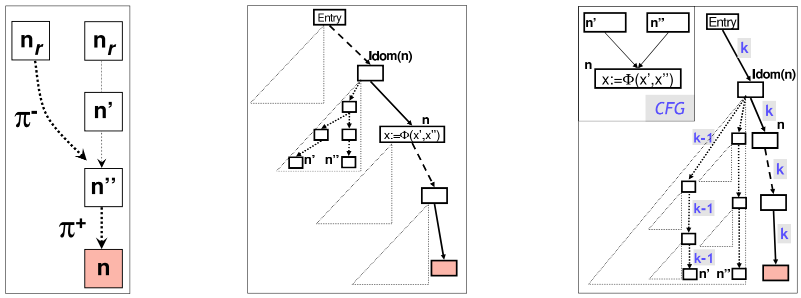

Fig. 2. (a) If a node $\mathtt{n}''$ not dominated by $\mathtt{n}'$ appears after the last occurrence of $\mathtt{n}'$ on a path to $\mathtt{n}$, then $\mathtt{n}$ is not dominated by $\mathtt{n}'$. (b) ϕ-strengthening: For a join ϕ-node $\mathtt{n}$, the strengthening $\Gamma(\mathtt{n})$ is the disjunction of the SIs of the two predecessors $\mathtt{n}', \mathtt{n}''$ of $\mathtt{n}$, which are in the tree "hanging off" $\mathsf{Idom}(\mathtt{n})$ (c) To compute the $(k-1)$-SI for $\mathtt{n}', \mathtt{n}''$ we strengthen all the join nodes in the path from $\mathtt{n}', \mathtt{n}''$ to the root $\mathsf{Idom}(\mathtt{n})$, recursively exploring the trees hanging off the inner paths.

i.e., the nodes that dominate the target node. The 2-SI strengthens the node predicates *for each join ϕ-node* $\mathtt{n}'$ along the path to the root in the dominator tree. To do so, it takes the disjunctions of the 1-SI for the predecessors of the join node. As shown in the figure, for join nodes, the predecessors are guaranteed to be in the subtree "hanging off" the join node's immediate dominator. Hence, the recursive SI for the predecessors is computed using the subtree rooted at the immediate dominator of the join node. The 3-SI would further strengthen each ϕ-node appearing in the recursive strengthening and so on. Thus, by increasing k we pick up more and more of the CFG nodes, but each node only appears once in the SI.

Example 4. Consider the ϕ-node $\mathtt{n}_2$ in the CFG of Figure 1(b). It is a join point and its two predecessors are the nodes $\mathtt{n}_1$ and $\mathtt{n}_{0'}$. Notice that in the dominator tree in Figure 1(c), the predecessors belong in the subtree hanging off the immediate dominator of $\mathtt{n}_2$ namely the entry node. We recursively compute the SIs: $\Psi(\mathtt{n}_1) = \mathtt{ip} \rightarrow \mathtt{check} = 0 \wedge \mathtt{ip} \rightarrow \mathtt{check}' = 1$ (from the dominators $\mathtt{n}_0, \mathtt{n}_1$), and, $\Psi(\mathtt{n}_{0'}) = \mathtt{ip} \rightarrow \mathtt{check} \neq 0$, (from the dominating branch condition $\mathtt{n}_{0'}$). Thus, the strengthening the ϕ-node $\mathtt{n}_2$ yields the following 2-SI for $\mathtt{n}_5$:

$$
\begin{array}{lll}
(\quad (\mathtt{ip} \rightarrow \mathtt{check}'' = \mathtt{ip} \rightarrow \mathtt{check}' & & \\
\quad \wedge\ \mathtt{ip} \rightarrow \mathtt{check} = 0 & \mathtt{n}_0 & \\
\quad \wedge\ \mathtt{ip} \rightarrow \mathtt{check}' = 1) & \mathtt{n}_1 & \\
\vee & & \Gamma(\mathtt{n}_2) \\
\quad (\mathtt{ip} \rightarrow \mathtt{check}'' = \mathtt{ip} \rightarrow \mathtt{check} & & \\
\quad \wedge\ \mathtt{ip} \rightarrow \mathtt{check} \neq 0)) & \mathtt{n}_{0'} & \\
\wedge\ \mathtt{t} = \mathtt{ip} \rightarrow \mathtt{proto} \wedge\ \mathtt{t} = \mathtt{TCP} & & \mathtt{n}_3 \text{ and } \mathtt{n}_4
\end{array}
$$

which is strong enough to prove the (implicit) assertion that the `check` field is non-zero, at the access location 5. A similar sufficient SI is obtained for 7. ∎

Example 5. **[Conditional Locking]** Figure 3(a) shows conditional locking on an arbitrary predicate p. Consider the ϕ-node $\mathtt{n}_4$ in the CFG of Figure 3(b).

```
example2() {
1:lock := 0;
2:if (p) {
3: lock := 1;
  }
4:
5:if (p) {
6: assert(lock=1);
   lock := 0;
  }
}
```

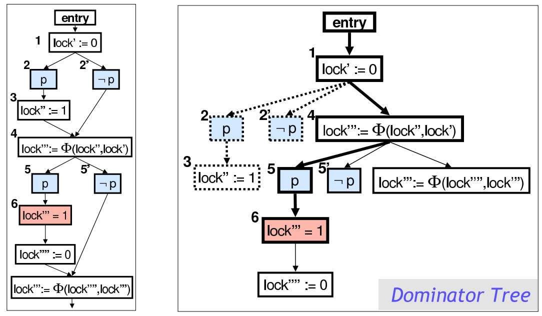

Fig. 3. (a) Example 2 (b) CFG (c) Dominator Tree

It is a join point and its two predecessors are the nodes $\mathtt{n}_3$ and $\mathtt{n}_{2'}$. Notice in the dominator tree in Figure 3(c), that the predecessors belong in the subtree "hanging" off the immediate dominator of $\mathtt{n}_4$ namely $\mathtt{n}_1$. We recursively compute the SIs: $\Psi(\mathtt{n}_3) = \mathtt{lock}' = 0 \wedge \mathtt{p} \wedge \mathtt{lock}'' = 1$ and $\Psi(\mathtt{n}_{2'}) = \mathtt{lock}' = 0 \wedge \neg \mathtt{p}$. Thus, the strengthening for the ϕ-node 4 is $\Gamma(\mathtt{n}_4) \equiv (\mathtt{lock}''' = \mathtt{lock}'' \wedge \Psi(\mathtt{n}_3)) \vee (\mathtt{lock}''' = \mathtt{lock}' \wedge \Psi(\mathtt{n}_2))$. We need not further strengthen the SIs for $\mathtt{n}_3, \mathtt{n}'_2$ as they have no dominating join nodes. The 2-SI at $\mathtt{n}_6$ is:

$$\begin{array}{ll} \quad \mathtt{lock}' = 0 & \text{from } \mathtt{n}_1 \\ \wedge\, ((\mathtt{lock}''' = \mathtt{lock}'' \wedge \mathtt{lock}'' = 1 \wedge p) \vee (\mathtt{lock}''' = \mathtt{lock}' \wedge \neg p)) & \text{from } \Gamma(\mathtt{n}_4) \\ \wedge\, p & \text{from } \mathtt{n}_5 \end{array}$$

This is an invariant strong enough to prove the assertion $\mathtt{lock}''' = 1$ at line 6. ∎

2.2 Interprocedural Structural Invariants

We now extend programs to include function calls. The set of operations is extended to include *function calls* $\mathtt{l} := f(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ and return statements $\mathtt{return}(\mathtt{ret})$, where $\mathtt{ret}$ is a special variable. A program is now a set of CFG's, one for each function, with a specified function `main` where execution starts. Further, we assume that the only operation on the exit node $\mathtt{n}_x$ of each CFG is $\mathtt{return}(\mathtt{ret})$, and the operation $\mathtt{return}(\cdot)$ does not appear anywhere else. We assume for simplicity there are no global variables, these can be incorporated with additional notation (and are handled by our implementation). We extend k-structural invariants to programs with function calls through two approaches: summarization and abstract summarization.

Summarization. For interprocedural analysis, each function is abstracted into a set of input-output relations, called the *summary*, that captures the observed behavior of the function. For function `foo`, we have to consider both transitive callees of `foo` (*i.e.,* calls to functions within the body of `foo`), and transitive callers of `foo` (*i.e.,* the call chains from `main` to `foo`).

To deal with callees, we extend $[\![\mathsf{op}]\!]$ to the new operations. First, assume there is no recursion. Let f be a function with formal parameters $\mathtt{x}_1, \ldots, \mathtt{x}_n$, local variables L, and CFG $G_f = (N^f, E^f, \mathtt{n}_e^f, \mathtt{n}_x^f)$. We define $[\![\mathtt{l} := f(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]$ as

$$(\exists L.\Psi((\mathtt{n}_e^f, \mathtt{n}_x^f), k))[\mathtt{l}/\mathtt{ret}, \mathtt{e}_1/\mathtt{x}_1, \ldots, \mathtt{e}_n/\mathtt{x}_n] \tag{1}$$

and $[\![\mathtt{return}(\mathtt{ret})]\!] = \mathit{true}$. Intuitively, we recursively construct the k-SI for the exit node of f, rename all local variables of f with fresh names (to avoid name clashes), and substitute the formal parameters and return variable in the expression. This k-SI is the summary of f. In the presence of recursion, we additionally pass the stack of function calls in the computation of $[\![\cdot]\!]$, and return $[\![\mathtt{l} := f(\mathtt{e}_1, \ldots, \mathtt{e}_n), s]\!] = \mathit{true}$ if f appears in the stack s.

To deal with callers, we generalize our definition of dominators to the interprocedural case, using the call graph of the program. In particular, we add edges from every call site $\mathtt{x} := f(\ldots)$ to the entry node $\mathtt{n}_e$ of f (but not edges from the exit nodes to the call sites), and compute dominators in this expanded graph. If $\mathtt{n}'$ dominates $\mathtt{n}$ in this expanded graph, then every return-free path from the entry node of `main` to $\mathtt{n}$ passes through $\mathtt{n}'$ (if $\mathtt{n}'$ and $\mathtt{n}$ are in the same function, we get back the original definition). The algorithm to compute k-SI for the transitive callers is then identical to the intraprocedural algorithm with this new definition.

Abstract Summarization. In abstract summarization, summaries are computed relative to two non-empty sets of *input* and *output* predicates for each function. Fix a function f. Let P and P' be the input and output predicates over variables in scope in f respectively. An abstract summary S is a subset of $P \times P'$ with the property that for every execution of the function starting from a state satisfying p to a state satisfying p', we have $(p, p') \in S$.

To perform abstract summarization, we traverse the call graph of the program bottom up. For any k, function f, and sets P and P' of predicates, our summarization algorithm constructs the k-SI φ of the exit node $\mathtt{n}_x^f$ of f with respect to the entry point $\mathtt{n}_e^f$ of f. For any function call $\mathtt{l} = g(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ in the body of f with summary S_g, we use the operation predicate from Equation 1 with $\Psi((\mathtt{n}_e^f, \mathtt{n}_x^f), k)$ replaced with $\bigvee_{(p,p') \in S_g}(p \wedge p')$. If g has not been summarized, *e.g.* , for recursive calls, we use the constraint *true*. Let φ be the k-SI for f. Finally, the abstract summary S_f of f is computed as the set $\{(p, p') \in P \times P' \mid p \wedge \varphi \wedge p' \text{ is satisfiable}\}$.

If $\bigvee P$ and $\bigvee P'$ are not both equivalent to *true*, abstract summarization can lead to unsoundness. To be sound, we add additional assertions to the program. At each call site $\mathtt{x} := f(e_1, \ldots, e_n)$, we add the assertion $\mathtt{assert}(\exists L. \bigvee P)[e_1/\mathtt{x}_1, \ldots, e_n/\mathtt{x}_n]$ which checks that the precondition of the function holds at the call site. At the exit node of f, we add the assertion $\bigvee P'$ that checks that the postcondition of the function holds at the return point. These assertions are checked in addition to the assertions in the program, and the original assertions are proved soundly if all these assertions also hold. If these assertions do not hold, the summary for the function is replaced with $(\mathit{true}, \mathit{true})$ when checking other assertions.

Abstract summarization allows our algorithms to scale by keeping the summaries small (just in terms of the abstract predicates), and also acts as a useful fault localization aid in our experiments. However, it requires user-supplied predicates, reducing automation. Instead of requiring user intervention or performing predicate inference [2, 21], we adopt the approach of [11, 24]. We perform abstract summarization with respect to predicates obtained automatically from the property. For example, to checking correct locking, we add predicates corresponding to each value of lock being taken or freed. This allows our tool to be automatic, though sometimes with less precision.

3 Experiments

We have implemented psi, an assertion checker for C programs using structural invariants. Our tool takes as input a C program annotated with assertions and a number k, statically constructs the k-structural invariant for each assertion, and checks if the k-structural invariant implies the assertion. Our tool is written in Objective Caml and uses the CIL library [22] for manipulating C programs. To prove an assertion, psi checks if the k-SI implies the assertion using the Simplify theorem prover [12]. The implementation is staged in five parts: alias analysis, SSA conversion, dominator tree construction, construction of the k-SI, and assertion verification. Our tool uses a flow-insensitive may alias analysis. After alias analysis, we transform the program so that conditionals on possibly aliased objects are added at each pointer dereference. This accurately reflects state update for the structural invariant. For example, for the code `*p = 5`, assuming `p` may point to `a` or `b`, we transform the code to `*p=5; if(p==&a) a=5; if (p==&b) b=5;`. Our alias analysis is field insensitive. We heuristically add field sensitivity based on field types to determine a more precise match. We ran three sets of experiments with psi: checking tagged unions, correct locking, and correct suid privileges. Our experiments were all run on a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. The running time is dominated by the alias analysis and the generation of the structural invariants. In comparison, the parsing, ssa conversion, dominator tree construction, and theorem prover calls take relatively little time.

1. Tagged Unions. Tagged unions are checked by adding an assertion describing the predicate that must hold when a certain field is accessed or cast before that access or cast. We added these assertions manually. We ran our tool on three programs: `icmp` (a protocol for error notification on the internet, 7K lines of code), `gdk` (the GTK+ drawing toolkit, 16K lines of code), and `lua` (an interpreter, 18K lines of code). We checked 69 assertions and found 14 false positives with $k = 2$ and 18 false positives with $k = 1$. The total run time was 684s with $k = 2$, dominated by `lua` (682s). Since most programmers check the tag near the data access point, we did not propagate k-SIs to the callers of the function containing the assertion for this set of experiments. This resulted in 8 false positives that required assumptions about formal parameters. Our theorem prover only models integers so there was 1 false positive that required the modeling of unsigned integers. Four false positives are due to modeling pointer arithmetic

program	LOC	func's	asserts	total	ok	error	ptrs	List	loops	unc	t(s)	cqual
`scc`	16K	638	36	57	47	2	7	0	0	1	38	60
`DAC960`	24K	763	46	54	38	0	10	0	4	2	141	N/A
`af_netrom`	22K	958	23	25	21	0	0	3	1	3	12	20
`af_rose`	23K	958	15	29	28	0	0	0	0	1	7	9
`as-iosched`	14K	576	10	17	10	0	4	0	0	3	8	4
`elevator`	13K	512	2	3	3	0	0	0	0	0	1	0
`floppy`	18K	696	30	48	43	0	0	0	2	3	35	48
`genhd`	13K	529	4	6	6	0	0	0	0	0	2	0
`ll_rw_blk`	15K	625	8	30	25	0	0	0	2	3	8	N/A
`nr_route`	18K	788	19	34	30	0	0	1	1	2	9	20
`wavelan_cs`	17K	621	19	35	30	1	4	0	0	0	14	4
`rose_route`	42K	953	51	73	55	13	0	0	3	2	35	31
Totals	235K	8,617	263	414	336	16	25	4	13	20	310	196

Table 1. Lock experiments. LOC is lines of code. Asserts gives the original number of asserts, and total gives the total asserts to check pre- and post-conditions. ok gives the asserts proved safe. error the number of bugs. False positives are broken into pointers (ptrs), lists, loops, and unclassified errors (unc). t(s) is time in seconds. Cqual shows the false positives from Cqual (N/A indicates we did not run Cqual).

and data structures and one due to type-unsafe programmer assumptions about memory layout.

2. Locking. The second set of experiments checked double locking errors in the Linux kernel. Double locking has been extensively studied using dataflow analysis [19] and BMC [24]. Double locking occurs when locking something that already has been locked (causing a deadlock) or unlocking something that already has been unlocked (can cause kernel panic). We model this by adding an assertion that the lock is in a locked (resp. unlocked) state before every call to `unlock` (resp. `lock`). We use abstract summarization for locks, similar to Saturn [24]. Predicates for abstract summarization are the lock values. Instead of user provided pre- and post-conditions, we use a simple heuristic to guess predicates and psi automatically checks whether those predicates are correct. For each function, we find the first assertions for each lock and make these the precondition predicates. Similarly, we find the last lock or unlock statements in the function and make the corresponding lock states the postcondition predicates. This is sometimes imprecise, but retains automation. We run psi with a depth $k = 2$. We found increasing $k > 2$ does not reduce the number of false positives in our experiments since a depth $k = 2$ captures all the relevant nesting of conditionals.

Table 1 summarizes our results. We examined 12 device driver files in the Linux kernel, totaling 235K lines of code. Since we consider drivers one file at a time, our current experimentation is unsound in the way we deal with function summaries. In particular, we do a global alias analysis at a per file level, but assume functions in other files do not have any effect on lock values or aliasing. There were a total of 414 asserts. Among these, 151 assertions were due to adding pre- and post-condition assertions for the summaries. We analyzed a total of 8,617 functions in 310 seconds. We found 16 real bugs and 62 false positives. We found errors in wavelan and rose_route not mentioned in the Saturn bug database. The bug in `wavelan_cs`, a wireless card driver, was caused by an obscure case where a packet is recieved when the wireless connection is being handed over from one access point to another. This bug spans 3 functions.

The false positives are in three categories: loss of precision in abstract summarization (25), getting locks from dynamic data structures or external functions (4), and loops (13). There are 20 additional errors we have not classified yet. Loop false positives occur when a lock is acquired and released in a loop. Interestingly, some such examples can be proved using Cqual or dataflow analysis, showing the orthogonality of these methods. Other false positives relate to dynamic data structures (where locks are stored in lists) or pointers returned from external functions.

Imprecise summary predicates are the most significant false positives (25 of them), but could be remedied by better predicate generation heuristics or some editing of the source code. When we heuristically add preconditions and postconditions, it is possible that the predicate we include in our precondition mentions a variable that is not in the formal parameter of our function or a global variable. For example, for the code

```
void lock (dev *ptr) {
  struct receive_queue *q;
  q = ptr -> q; assert (q -> lock == 0); q -> lock = 1; }
```

our heuristics infer that the pre- and post-conditions are `(q->lock == 0)` and `(q->lock == 1)` respectively. This can be solved by correcting the pre- and post-conditions to `(ptr->q->lock == 0)` and `(ptr->q->lock == 1)` respectively. The 25 false positives involving these issues are all removed after these simple modifications. Alternately, we could construct the weakest precondition of these predicates in terms of the formals and used those for abstract summarization.

To compare, we ran Blast [21] on some of the Linux drivers. Table 2 summarizes the results of running Blast on three of the drivers. While the false positive rate is lower (although not zero, since Blast produces false positives when locks are put into lists, and when the driver makes unmodeled assumptions about external pointers), the time taken is significantly higher in all cases.

Program	FP	Time (s)
`af_netrom`	4	248
`nr_route`	1	1755
`rose_route`	1	1513

Table 2. Blast results

Our work in lock analysis is most similar to Saturn. We found all bugs that Saturn found except one that required analysis of two different files. In addition, we find two extra bugs not reported by Saturn. In comparison with Cqual [19], we take more time, but have fewer false positives. Running Cqual on 10 of our 12 device driver examples resulted in 196 type errors, even though Cqual does reduce loop false positives. In contrast to Cqual, we get at most one message per assertion site, making it easier to track down false errors.

Finally, Table 3 summarizes the precision-time trade-off as we increase k over all our lock experiments. Size measures the total size of the Simplify queries written as a text file, time is the time to solve all the queries, and FP the number of false alarms found. In our experiments, $k = 1$ is already enough to prove most assertions, and increasing k beyond to 2 does not help in reducing the false alarms. The size of the formula does not increase appreciably beyond $k = 4$. For our examples, it is rare to find complex control flow, *i.e.,* more than four nested conditionals.

k	Size (KB)	Time	FP
1	259	1.5s	241
2	15764	1m45s	62
3	19996	1m54s	62
4	21091	1m58s	62

Table 3. Precision

3. Privilege Levels. Finally, we checked whether a Unix setuid program gives up its owner privileges before executing certain system calls [7]. In unix systems, programs have privileges associated with users. Normally, a program will execute under the permission of the user executing the program. However, *suid programs* run with root privileges when they are started, which are required to access certain system resources. After the privileged action is performed, suid programs give up their root privileges by making a `setuid` or a `seteuid` call. A suid program should give up its root privileges before making further system calls to reduce the chance of an exploit gaining root access. We model the effective user id with an integer which is 0 for root, and 1 for any other user. The id is set to 1 whenever `setuid` or `seteuid` is called. We check that whenever a program calls `system` or `exec`, this id is not zero. We examine three programs: OpenSSH 2.9.9p (the widely used secure shell program), GNU Privacy Guard (open source pgp), and mtr (a network diagnostic tool), for a total of 294K lines of code. All these programs follow good security programming guidelines. After the required privileged action was taken, the effective user id was set to the user executing the program. We used abstract summarization, using the state of the id bit as the predicate. This caused 254 out of 270 assertions to be automatically added, however, summarization made our technique scale well. Further, $k = 1$ was enough to prove all assertions with no false positives. Our results are shown in Table 4, where the time does not include time for alias analysis. Our total running time (excluding alias analysis) was 20 minutes. As the size of the programs increased, CIL's alias analysis became the bottleneck. It took 610s for OpenSSH and did not terminate for gpg within 6 hours. However, since the address of the id bit is not taken and id is only assigned integer constants, and we additionally check that the k-SI is satisfiable, we can conclude in this case that our technique is sound without the alias analysis. In comparison, Blast did not finish the verification of openssh or gpg in two hours.

4 Related Work and Conclusions

Related Work. SIs are similar to bounded model checking (BMC) [4, 9, 24], which builds VCs capturing all program executions of a certain bounded execution length. Typically BMC is useful for finding bugs, while SI provides a sound verification technique. While BMC unrolls the last (or first) k operations of a

Program	LOC	Asserts	Original	FP	Time (s)
mtr	13K	43	8	0	13s
openssh	61K	37	5	0	51s
gpg	219K	190	3	0	1106s

Table 4. Suid Programs. Asserts = total number of asserts, Original = original asserts in the code, FP = false positives.

program, the "unrolling metric" in k-structural constraints is (roughly) the nesting depth of conditionals. Thus, 1-SI may be strong enough to prove a property even though the relevant code blocks are separated by arbitrarily many lines of irrelevant code. SIs are less precise than VC based program verification tools [16], but we have demonstrated that the loss of precision is not significant for a large class of interesting properties. We have traded off precision for automation and scalability. Algorithms for computing compact weakest preconditions have been studied [17, 3], however these did not consider the effects of approximating the VC using the nesting depth, and the results of the loss of precision in property checking.

Counterexample-guided abstraction refinement [8, 2, 21] automates the discovery of abstractions using spurious counterexamples. While theoretically as efficient as SI and as complete as general VC-generation, in practice, these tools do not scale well for large programs even if there is an "obvious" proof of correctness. This is mainly because these tools strive to be generic, and do not always exploit "simple" control/data flow tricks, reverting to more expensive but more general symbolic processing. In fact, our motivation for this work was the observation that simple algorithms can filter out many assertions quickly before these more sophisticated tools are applied.

SSA and dominators have been used to find program invariants that facilitate certain compiler optimizations [1, 6] and to check security properties [25]; our work is a generalization of these algorithms to arbitrary invariants. Independent of our work, dominator invariants have been recognized as a quick way to generate invariants for translation validation [15]. However, that work does not provide a parameter to adjust the precision.

SI vs Dataflow Analysis. Another scalable technique of finding invariants is via fixpoint computations over an abstract domain of dataflow facts tailored to the property being checked. Examples are [5, 23], which use sophisticated domains to find complex invariants over data, or [19, 14, 11] which address more control-oriented properties. Our VC-based method provides a scalable and generic technique to introduce path correlations incrementally to a variety of simple properties without requiring an expert-specified and program dependent abstract domain.

The invariants obtained using k-SI and (flow-sensitive) dataflow analysis that merges information at join points are, in general, incomparable. For $k > 1$, the k-structural constraints incorporate path correlation information that dataflow analysis merges. For $k = 1$, if the domain of dataflow facts is fixed (as is usual) from the property and not tailored to a particular program, the dominator invariant may be more precise. For example, suppose $\mathtt{p} = \mathtt{p}_1 \wedge \mathtt{p}_2$, and consider the program:

```
if (p1) { if (p2) { L: assert(p); } }
```

where the dataflow domain only tracks `p` (obtained from the assert).

On the other hand, there are programs where dataflow analysis is more precise. Consider:

```
x := 1; while (*) { if(x=1) x := 1; } L: assert(x=1);
```

When the state of x is tracked, a dataflow analysis produces the invariant $x = 1$ at `L`. However, for any k, the k-SI at `L` is *true*, since x within the loop is unconstrained.

Conclusions. SIs form a scalable, lightweight algorithm to prove useful properties of programs. Although our algorithm is simple, we showed it can prove many instances of useful and well-studied properties such as setuid, locking, and tagged unions. These programs and properties are frequently used to test more complex tools such as SLAM or Blast. However, in our experience, for the same properties and programs, Blast is usually an order of magnitude slower than psi. even though the false positive rate is only slightly better than SIs in the programs and properties we checked.

Thus, we advocate a hybrid verification approach where efficient, simple tools that incorporate structural idioms are run first to eliminate most assertions, and more sophisticated but slower tools are focused on the remaining assertions that escape the purview of the simple tools.

References

1. B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *POPL 88*, pages 1–11. ACM, 1988.
2. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.
3. M. Barnett and K.R.M. Leino. Weakest-precondition of unstructured programs. In *PASTE 2005*, pages 82–87. ACM, 2005.
4. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 99: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 193–207. Springer, 1999.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI 03: Programming Languages Design and Implementation*, pages 196–207. ACM, 2003.
6. R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI 00*, pages 321–333. ACM, 2000.
7. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *NDSS 04: Annual Network and Distributed System Security Symposium*, pages 171–185, 2004.

8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer, 2000.
9. E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS 04: Tools and Algorithms for the construction and analysis of systems*, LNCS 2988, pages 168–176. Springer, 2004.
10. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadek. Efficiently computing static single assignment form and the program dependence graph. *ACM Transactions on Programming Languages and Systems*, 13:451–490, 1991.
11. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, 2002.
12. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
13. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
14. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI 00: Operating System Design and Implementation*. Usenix Association, 2000.
15. Y. Fang. *Translation validation of optimizing compilers*. PhD thesis, 2005.
16. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.
17. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL 00: Principles of Programming Languages*, pages 193–205. ACM, 2000.
18. R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.
19. J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02: Programming Language Design and Implementation*, pages 1–12. ACM, 2002.
20. D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, 2003.
21. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
22. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, LNCS 2304, pages 213–228. Springer, 2002.
23. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
24. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL 05: Principles of Programming Languages*, pages 351–363. ACM, 2005.
25. X. Zhang, T. Jaeger, and L. Koved. Applying static analysis to verifying security properties, 2004. Grace Hopper Conference.