Assertion Verification using Structural Invariants

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

[1]CS Department, University of California, San Diego
[2]CS Department, University of California, Los Angeles

Abstract. We present a new assertion verification algorithm based on *structural invariants* (SI) computed by making a linear pass over the dominator tree of a program in SSA form. 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 joins of the 1-level SI with the $(k-1)$-level SI of the join's predecessors, thereby distinguishing k levels of nested conditionals. SI provide a sound, generic (no program-specific abstract domain required), scalable, and precise (low false positive rate) algorithm for path sensitive assertion verification. We have implemented our algorithm in psi, an assertion verifier for C programs. We have used psi to experimentally validate that for a large class of open source programs and properties —including tagged union, locking, and security properties— our technique provides a scalable algorithm with a low false positive rate, even with a nesting level $k \leq 2$.

1 Introduction

Verification-conditions(VC) are a classical technique for verifying properties of programs [**?**,**?**]. A VC at a program location is a (first-order) predicate that holds whenever the location is visited during execution. Thus to prove that a programmer specified predicate always holds at a location, it suffices to check if the VC implies the asserted predicate.[1]

The use of VCs has been limited 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 precisely capture all program executions, VC generators encode all execution paths of the program, resulting in large formulas, limiting their use on large programs. Thus, while *generic*, in the sense that they are applicable to any user specified assertion, and *precise*, as they are able to analyze path correlations, the use of VC-based techniques has been limited to proving deep properties of programs through extensive manual intervention.

For properties that must be checked 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

[1] Our use of the term VC differs from traditional use where the implication "predicate implies assertion" is called the VC.

over a fixed abstraction. These techniques often sacrifice genericity and precision to gain automation and scalability: they use property-specific abstractions to gain automation and thus are not generic, they gain scalability by merging execution paths at join points, thus ignoring correlated branching behavior, leading to imprecision in the form of false alarms.

In this paper, we present a VC generation technique that is automatic and scalable enough to prove many simple but useful safety properties over large code bases, without requiring an expert to devise a specialized analysis for each property. ♣ say here – no loop invs ?♣ 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, using 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.SIs use the dominator tree to capture control flow information and the SSA form to capture data flow information about the program.

The first and coarsest over-approximation (the 1-structural invariant) is obtained as the *conjunction* of the dominating operations' predicates. 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 the SI of the 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 linear time in the program size, to obtain a SI which is a boolean combination of the statements syntactically appearing in the program. For each $k > 0$, the k-SI provides an over-approximation of the VC, becoming more precise with increasing k.

In the limit (k = size of the CFG), the SI is equivalent to the standard VC [13] obtained by unrolling each loop of the program once, arbitrarily updating the loop-modified variables are havoced. Thus, SI are only effective for those properties for which the trivial loop invariant *true* suffices, but we demonstrate empirically, that for several key properties, for each of which, a specialized scalable analysis has been designed previously, the SI-based VC is precise enough to prove the property.

SI achieve scalability through the parameter k, which provides a tunable selector for statements that most influence the assertion. For example, 1-SI includes only the operations that must happen on *all* CFG paths to the target. Empirically, we have found the 1-SI to be two orders of magnitude smaller than a full VC that sweeps over the entire program. Thus, the resulting validity queries are discharged upto 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 the bulk of the remaining assertions, the 2-SI sufficed. We conjecture this is because there is enough path-sensitive control and data-flow information embedded in the dominator tree to prove a large class of useful assertions, even when variables modified in loops are left unconstrained. Clearly, there are programs and properties where this requirement does not hold, in which case the SI is too weak resulting in false alarms.

To validate our empirical observation, and thus demonstrate the precision and genericity of our technique, we have used SI to successfully analyze a diverse set of open-source programs for three important safety properties with a low false positive rate.

♣ property...is to verify doesnt sound right♣ The first property (studied in [**?**]) is to verify 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. The second property, (studied [15, 24, 11]) is to verify that Linux drivers acquire and release locks in strict alternation, studied in In most cases each call to `unlock` is dominated by a call to `lock` and vice versa. 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 (studied in [6, 5]) is to verify that at any point where a `suid` program calls `execv`, the effective user-id is non-root [6, 5]. It turns out that system calls setting the user-id dominate the call to `execv` and so the 1-SI suffices to prove the assertion.

We have built a tool psi, that generates VCs incrementally using structural-invariants, and uses them to verify programmer specified assertions. psi takes as input a C program annotated with programmer assertions, and a number k, and computes the k-SI for the program at each assertion point, and then uses SIMPLIFY [**?**] to discharge the validity query, and thus prove the assertion. We have used psi to check the properties described above on a total of 570K lines of code containing 759 assertions. ♣ how many proved ? how many bugs? how many false-alarmes ?♣ The total running time of all experiments was less than one hour.

Related Work. Notice that SIs are dual approximations of the (unsound) bounded model checking (BMC) technique [2, 7, 19, 24], which builds a VC capturing all program executions of a certain bounded execution length. While BMC is very useful for finding bugs, SI provides a *sound* verification technique. Further, while traditional BMC unrolls the last k operations of a program, the "unrolling metric" in k-structural constraints is (roughly) nesting depth of conditionals. For example, a choice of $k = 1$ may be enough even to prove a property even though the relevant code blocks are separated by arbitrarily many lines of code. ♣ fix this so that we partition between tools that do deep things TVLA and those that do not eg FTA02, Engler00♣. Another classical way to obtain precise and scalable analyses is by using a fixpoint computation to find invariants expressible using a

specified abstract domain [3, 23], or dataflow facts [15, 11, 9]. [5, 21]. Each of the above tailors the abstract domain to the properties checked: the abstract domain preserves enough information at join points to make the analysis precise for the properties of interest, and throws away enough information to make the tools scalable. Unfortunately, the tailoring to the abstract domain means the tools are not generic. Counterexample-guided refinement generalizes this by refining the abstract domain using false positives (spurious counterexamples) [1, 17, 4]. While generic, so far their success has been limited to small, control-dominated applications such as device drivers.

2 Example

We now illustrate structural invariants by showing how they can be used to check two kinds of properties: tagged unions and correct use of locks in device drivers.

Property 1: Tagged Unions. The first problem we consider is that of checking correct usage of union types in C programs, where the unions are not tagged explicitly by the language. C programmers use a *tag field* to determine the type of the union instance. Thus, the value of the tag field is checked and the data from the union cast 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.

Example 1. Figure 1(a) shows an example of using tagged union. The code 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. 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: $\texttt{ip} \rightarrow \texttt{proto} = \texttt{TCP}$ and one at line 7 where `h` is cast to `UDP` pointer which asserts: $\texttt{ip} \rightarrow \texttt{proto} \neq \texttt{TCP}$. To check correct usage of tagged unions, we must check these assertions hold for every program execution path to those assertions.

Verifying assertions using Invariants. Internally, we represent a program as a control flow graph where nodes correspond to basic blocks of assignments or assume predicates representing conditionals. The *assertion verification* problem then asks if an assertion φ holds at control flow node `n` whenever control reaches `n`. A classical way to check this is to demonstrate the existence of an `n`-invariant, *i.e.,* a predicate over the program variables that holds after every execution path ending at `n`, that is stronger than the assertion.

Dominator Invariants. Our first cut at computing invariants is through node dominators. A control-flow node `n` is *dominated* by another node $\texttt{n}'$ if *all* execution paths to `n` must first pass through $\texttt{n}'$. The dominator relationship of a CFG can be efficiently computed [20] and succinctly represented using a *dominator*

tree, where a node $\mathtt{n}'$ is an ancestor of $\mathtt{n}$ if it is a dominator of $\mathtt{n}$. We use the nodes (operations) that dominate $\mathtt{n}$ to efficiently compute $\mathtt{n}$-invariants by making the following observations.

1. Immediately after the operation of a node has been executed, a predicate corresponding directly to the operation holds of the program's state. That is, if the operation was $\mathtt{x} := \mathtt{e}$, then immediately after the assignment, the program satisfies the predicate $x = e$ (assuming $\mathtt{x}$ does not appear in $\mathtt{e}$). We call this predicate the *operation predicate* of $\mathtt{n}$.
2. If $\mathtt{n}'$ dominates $\mathtt{n}$, then every execution leading to $\mathtt{n}$ *must* pass through $\mathtt{n}'$. If the variables appearing in the operation predicate of $\mathtt{n}'$ are *never modified* after control passes through $\mathtt{n}'$, then the operation predicate of $\mathtt{n}'$ is a $\mathtt{n}$-invariant. All executions to $\mathtt{n}$ pass through $\mathtt{n}'$. Immediately after executing $\mathtt{n}'$, the operation predicate of $\mathtt{n}'$ holds. As none of those variables are subsequently modified, the operation predicate of $\mathtt{n}'$ is preserved and holds after execution to $\mathtt{n}$.
3. The additional stipulation mentioned above does not hold in general. However, converting the program to *Static Single Assignment* (SSA) [8] form suffices to enforce it. Once the program is in SSA form, the variables *assigned to* in $\mathtt{n}'$ are are only assigned to *at* $\mathtt{n}'$, and thus, are not modified after the *last occurrence* of $\mathtt{n}'$ along the path to $\mathtt{n}$. We observe that SSA form additionally ensures that the variables *read* at $\mathtt{n}'$ are not modified after the *last occurrence* of $\mathtt{n}'$ along the execution to $\mathtt{n}$.

Together, these imply if $\mathtt{n}'$ dominates $\mathtt{n}$, then the operation predicate of $\mathtt{n}'$ is an $\mathtt{n}$-invariant. As the conjunction of $\mathtt{n}$-invariants is a $\mathtt{n}$-invariant, we get our first result: the predicate $\Psi(\mathtt{n})$, which is the conjunction of operation predicates for all the operations dominating a node $\mathtt{n}$, is a $\mathtt{n}$-invariant. We call $\Psi(\mathtt{n})$ the *dominator invariant* for $\mathtt{n}$.

Example 2. **[Dominator Invariants]** 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 $\mathtt{n}_5$ are those in the path to the root (shown in solid black), namely the nodes $\mathtt{n}_2, \mathtt{n}_3, \mathtt{n}_4$. By conjoining their respective operation predicates of the nodes 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 at 5: $\mathtt{ip} \rightarrow \mathtt{proto} = \mathtt{TCP}$. By virtue of the program being in SSA form, the dominator invariant captures the flow of value through the local variable $\mathtt{t}$.

Property 2: Double Locking. The second verification problem we consider is that of checking that locks are acquired and released correctly in kernel code. In particular, for each lock, calls to `lock` and `unlock` must alternate, otherwise the kernel may deadlock or panic [15, 11]. Thus, at each node where `lock` (respectively `unlock`) is called, there is an implicit assertion that the lock is *not* held (respectively is held).

Example 3. An instance of code where this property is hard to verify is shown in Figure 2(a). For brevity, we assume there is a single variable `lock`. Calls to `lock`(resp. `unlock`) are modeled by assigning `1` (resp. `0`) to `lock`. For clarity, we have explicated the unlock assertion in the figure. Instances like these are a notorious source of false positives, as path-sensitivity is needed to correlate the two branches. In Figure 2(b), we show the CFG for the code, after transformation to SSA form. From the dominator tree shown in Figure 2(c), we see that the nodes dominating $\mathtt{n}_6$ are $\mathtt{n}_1, \mathtt{n}_4, \mathtt{n}_5$, whose operation predicates are respectively: $\mathtt{lock}_1 = 0$, $\mathtt{lock}_3 = \mathtt{lock}_1 \vee \mathtt{lock}_3 = \mathtt{lock}_2$, and p. Though the conjunction of the operation predicates is a $\mathtt{n}_6$-invariant, it is not strong enough to prove the assertion $\mathtt{lock}_3 = 1$. This is because $\mathtt{lock}_2$ which may be assigned at the ϕ-node (join) $\mathtt{n}_4$ is unconstrained, as it is defined in $\mathtt{n}_3$ which does not dominate the target node $\mathtt{n}_6$. Thus, dominator invariants do not capture branch correlations. As they only conjoin operation predicates for dominating operations, They fail to capture the correlation that if p holds, then in the ϕ node, the definition of `lock` came from $\mathtt{lock}_2$, which, together with the definition of $\mathtt{lock}_2$ is sufficient to prove the assertion.

k-Structural Invariants. In order to introduce path sensitivity, for any integer $k > 0$, we generalize dominator invariants to *k-structural invariants* (k-SI) by distinguishing at join points (ϕ-nodes) the constraints that hold for the variables merged along the different predecessors.

Consider any ϕ-node $\mathtt{n}'$ that dominates the target node $\mathtt{n}$. We show that if neither predecessor of $\mathtt{n}'$ is dominated by $\mathtt{n}'$, (*i.e.,* if the ϕ-node $\mathtt{n}'$ is not a loop head), then, since execution must have come through one of the two predecessors, the *disjunction* of the dominator invariants of the predecessors is an $\mathtt{n}$-invariant. Moreover, each disjunct can be strengthened by constraining the value of the merged variable to be that which is held at the corresponding predecessor. More precisely, if $\mathtt{n}'$ is a ϕ-node with the assignment, $x := \phi(x', x'')$, without loss of generality, two predecessors $\mathtt{n}''$ and $\mathtt{n}'''$, neither of which is dominated by $\mathtt{n}'$, then we can *strengthen* the node predicate of $\mathtt{n}'$, by conjoining it with the predicate:

$$\Gamma(\mathtt{n}') \equiv (x = x' \wedge \Psi(\mathtt{n}'')) \vee (x = x'' \wedge \Psi(\mathtt{n}'''))$$

We show that instead of using the dominator invariants of the predecessors, we can recursively expand all the ϕ-nodes dominating them in the same manner, thus yielding a *Structural Invariant* (SI) for each node of the CFG. To prove that the predicate thus formed is indeed a $\mathtt{n}$-invariant, we generalize the reasoning for dominator invariants and prove by induction that the recursively computed predicates for the predecessors are indeed invariants for the predecessors, and that one of them must hold at the join point, and that it is subsequently preserved along the execution to $\mathtt{n}$, and hence the strengthening above is a $\mathtt{n}$-invariant.

Example 4. [**Strengthening ϕ-nodes**] Consider the ϕ-node labeled 4 in the CFG of Figure 2(b). 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 2(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.

By only recursively "splitting" on nodes that are not loop-headers, we ensure that this recursive process is guaranteed to terminate (in linear time). Thus, while we can compute the exact SI, the above technique allows us to build coarser approximations of it, depending on the depth of the recursion. That is, a 1-SI for a node is its dominator invariant, a k-SI for a node is its dominator invariant conjoined (strengthened) with the $(k-1)$-SI for the ϕ-node dominators that are not loop headers. In other words, a k-SI "unfolds" the *nesting structure* of the program. 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. Surprisingly, we find that for the diverse class of assertions we have verified, a branch-width of 2 suffices to prove the properties.

Example 5. **[2-Structural Invariants]** For the code of Figure 2, the 2-SI at $\mathtt{n}_6$ is:

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

For the ϕ node $\mathtt{n}_4$ corresponding to line 4, we have recursively computed the 1-SI (*i.e.,* the dominator invariant) for the two predecessors of the ϕ-node as described previously. We factor out the common dominator $\mathtt{n}_1$ from the disjuncts. Notice that this 2-SI is an invariant that is strong enough to prove the assertion $\mathtt{lock}''' = 1$ at line 6.

Example 6. **[Generalized Data Access Verification]** To illustrate the power of Structural Invariants, we revisit the packet processing example of Figure 1. In networking code, there is an additional requirement: the payload `h` can only be accessed after the checksum has been verified to ensure its integrity. The checksum verification "stamps" the `check` field, by setting it to `1`. The specification is that whenever the header field `h` is accessed, the value of the `check` field should be non-zero. This yields the additional implicit assertions at statements `5:` and `7:` that $\mathtt{ip} \rightarrow \mathtt{check}'' \neq 0$. Notice that the dominator invariant is not strong enough to prove this assertion. The 2-SI at the node `5` is the predicate:

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

which is strong enough to prove the implicit assertion. A similar sufficient SI is obtained for `7:`. In the above 2-SI, we strengthen the $\phi-$node $\mathtt{n}_2$ with the disjunctions of the dominator invariants of its two predecessors $\mathtt{n}_1, \mathtt{n}_{0'}$.

A Combinatorial Interpretation. A combinatorial interpretation for k-SIs is obtained by considering the (w.l.o.g. binary) CFG as a Markov chain where each outgoing edge is taken with probability $\frac{1}{2}$. For a node $\mathtt{n}$, the k-SI picks up all nodes $\mathtt{n}'$ in the CFG such that the conditional probability of visiting $\mathtt{n}'$ given a path in the Markov chain to $\mathtt{n}$ is greater than or equal to $\frac{1}{2^{k-1}}$.

SI vs Dataflow Analysis. The invariants obtained using k-SI and (flow-sensitive) dataflow analysis are, in general, incomparable. Clearly, for $k > 1$, the k-structural constraints incorporate path correlation information that dataflow analysis merges. However, even dominator invariants may produce more precise invariants: consider the program

```
if (p) { if (p) {} else {}; L: assert(p); }
```

where the predicate `p` is the dataflow fact being tracked. Since path correlations are not tracked, dataflow analyses will produce the invariant *true* (or "top") at statement `L`, since the then and the else branch of the inner if statement will be merged at `L`. However, a dominator invariant will correctly identify the outer `if` as a dominator, and produce the invariant `p`, which is more precise.

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

```
x = 0; while (*) { x = 1; x = 0; } L: assert(x=0);
```

When the state of x is tracked, a dataflow analysis produces the invariant $x = 0$ at `L`. However, for any k, the k-SI at `L` is *true*.

3 Syntax and Semantics

We formalize structural invariants for a simple imperative language with integer variables. We begin with the simpler intraprocedural case.

3.1 Syntax

Operations. Our programs are built using the operations:

1. *Assignment* operations $\mathtt{x} := \mathtt{e}$, which corresponds to assigning the value of expression `e` to the variable `x`. A *basic blocks* 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.

For an operation `op`, we write $\mathsf{wr}(\mathsf{op})$ for the set of variables written to by `op`. For a basic block $\mathtt{x}_1 := \mathtt{e}_1;\ \ldots;\ \mathtt{x}_k := \mathtt{e}_k$, we have $\mathsf{wr}(\mathsf{op}) = \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$, and

for an assume, $\mathsf{wr}(\texttt{assume}\ (\mathsf{p})) = \emptyset$. We write $\mathsf{vars}(\mathsf{op})$ to denote the set of all variables appearing syntactically in the operation op.

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

1. A set of control nodes N, each of which is 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 $\mathtt{n}$ in the CFG. We assume that the set $\mathsf{pred}(\mathtt{n})$ is ordered, and refer to the k-th predecessor of a node $\mathtt{n}$ to denote the k-th element in the ordering in $\mathsf{pred}(\mathtt{n})$. We write $\mathsf{wr}(\mathtt{n})$ and $\mathsf{vars}(\mathtt{n})$ for $\mathsf{wr}(\mathsf{op})$ and $\mathsf{vars}(\mathsf{op})$ for the operation op labeling node $\mathtt{n}$.

Paths. A *path* π of length m to a node $\mathtt{n}$ 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 $\mathtt{n}$ is *reachable* in the CFG if there is a path π to $\mathtt{n}$. We assume that all nodes in N are reachable from $\mathtt{n}_e$.

Dominators. For two CFG nodes $\mathtt{n}, \mathtt{n}'$ we say $\mathtt{n}$ *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 $\mathtt{n}$ *strictly dominates* $\mathtt{n}'$, written $\mathtt{n}\ \mathsf{D}\ \mathtt{n}'$ if $\mathtt{n}$ dominates $\mathtt{n}'$ and $\mathtt{n}, \mathtt{n}'$ are distinct. We write $\mathsf{D}(\mathtt{n})$ for the set $\{\mathtt{n}' \mid \mathtt{n}'\ \mathsf{D}\ \mathtt{n}\}$. We write $\mathsf{D}^{-1}(\mathtt{n})$ for the set $\{\mathtt{n}' \mid \mathtt{n}\ \mathsf{D}\ \mathtt{n}'\}$. We say $\mathtt{n}$ is the *immediate dominator* of $\mathtt{n}'$ if for every $\mathtt{n}'' \in \mathsf{D}(\mathtt{n})$, we have $\mathtt{n}''$ dominates $\mathtt{n}$. It is well known that each node $\mathtt{n}$ of the CFG has a unique immediate dominator which we write as $\mathsf{ID}(\mathtt{n})$. 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 $\mathsf{ID}(\mathtt{n})$.

SSA. We assume that programs are represented in *static single assignment* (SSA) form [8], in which each variable in the program is defined exactly once. Programs in SSA form have special ϕ-assignment operations of the form $\mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_\ell)$ 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, $\mid \mathsf{pred}(\mathtt{n}) \mid = n$, (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). We use the following facts about CFGs in SSA-form and their dominator trees.

Proposition 1. [Dominators and SSA-Form]

F1 *For every* $\mathtt{n}, \mathtt{n}' \in \mathsf{D}(n)$ *and path* π *to* $\mathtt{n}$ *of length* m*, there exists* $1 \leq i < m$ *such that (i)* $\pi(i) = \mathtt{n}'$*, and, (ii) for every* $i < j \leq m$*,* $\mathtt{n}'\ \mathsf{D}\ \pi(j)$*.*

F2 *If* $\mathtt{n}\ \mathsf{D}\ \mathtt{n}'$ *and* $\mathsf{wr}(\mathtt{n}') \cap \mathsf{vars}(\mathtt{n}) \neq \emptyset$*, then* $\mathtt{n}$ *is a* ϕ^ℓ*-node.*

F3 *(Def-Use Chain) If* $\mathsf{vars}(\mathtt{n}) \cap \mathsf{wr}(\mathtt{n}')$ *is not empty, then there is a path from* $\mathtt{n}'$ *to* $\mathtt{n}$ *in the CFG along which every intermediate node is dominated by* $\mathtt{n}'$*.*

F4 *If* $\mathtt{n} \notin \mathsf{D}^{-1}(\mathtt{n}')$ *and* $\mathsf{wr}(\mathtt{n}') \cap \mathsf{vars}(\mathtt{n}) \neq \emptyset$ *then* $\mathtt{n}$ *is a* ϕ^ℓ*-node or a* ϕ*-node.*

F5 *If* $\mathsf{wr}(\mathtt{n}') \cap \mathsf{vars}(\mathtt{n}) \neq \emptyset$*, and* $\mathtt{n}'$ *is a* ϕ*-node then* $\mathtt{n} \in \mathsf{D}^{-1}(\mathsf{ID}(\mathtt{n}'))$*.*

F6 *(Reducibility) In every cycle in the CFG there is a node that dominates all the other nodes of the cycle.*

3.2 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 that $s \overset{\mathsf{op}}{\rightsquigarrow} s'$ if:

$$s' = \begin{cases} s & \text{if } \mathsf{op} \equiv \mathtt{assume}\ (\mathtt{p}) \text{ and } s \models \mathtt{p} \\ s[\mathtt{x} \mapsto s.\mathtt{e}] & \text{if } \mathsf{op} \equiv \mathtt{x} := \mathtt{e} \end{cases}$$

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 $s \overset{\mathsf{op}}{\rightsquigarrow} s'$.

An alternative way to view the semantics of programs is via logical formulas. 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'.

Operation Predicates. The *operation predicate* of an operation, written $[\![\mathsf{op}]\!]$, is a predicate defined as:

$$[\![\mathsf{op}]\!] = \begin{cases} \mathtt{x} = \mathtt{e} & \text{if } \mathsf{op} \equiv \mathtt{x} := \mathtt{e} \\ \mathtt{p} & \text{if } \mathsf{op} \equiv \mathtt{assume}\ (\mathtt{p}) \\ \bigvee_{i=1}^{n} \mathtt{x} = \mathtt{x}_i & \text{if } \mathsf{op} \equiv \mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_n) \\ \mathit{true} & \text{if } \mathsf{op} \equiv \mathtt{x} := \phi^\ell(\mathtt{x}_1, \ldots, \mathtt{x}_n) \\ \bigwedge_{i=1}^{n} [\![\mathsf{op}_i]\!] & \text{if } \mathsf{op} \equiv \mathsf{op}_1;\ \ldots;\ \mathsf{op}_n \end{cases}$$

We extend the operation predicates to nodes $[\![\mathtt{n}]\!]$ which are defined as $[\![\mathsf{op}]\!]$ where op is the operation labeling $\mathtt{n}$. If the program is in SSA form, then the operation predicate $[\![\mathsf{op}]\!]$ is a postcondition of *true* w.r.t. the operation op. For a node $\mathtt{n}$ we write:

$$\Phi(\mathtt{n}, j) = \begin{cases} \mathtt{x} = \mathtt{x}_j & \text{if } \mathtt{n} \text{ is labeled } \mathtt{x} := \phi(\mathtt{x}_1, \ldots, \mathtt{x}_j, \ldots, \mathtt{x}_n) \\ [\![\mathtt{n}]\!] & \text{otherwise} \end{cases}$$

In other words, $\Phi(\mathtt{n}, j)$ constrains the variable assigned at a ϕ-node to the value that held at the j-th predecessor of $\mathtt{n}$.

Invariants. We say that *a path π satisfies the formula φ* if *after* executing the path π from any initial state, the state of the program satisfies φ. 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.

We shall use three properties about paths and node invariants. The first, **(I1)** is that if φ_1 and φ_2 are $\mathtt{n}$-invariants, then whenever execution reaches $\mathtt{n}$, both the predicates are satisfied, and hence, $\varphi_1 \wedge \varphi_2$ is a $\mathtt{n}$-invariant. The second, **(I2)** is that if the program is in SSA form, then along any path, if op was the last operation performed along the path, then the path satisfies $[\![\mathsf{op}]\!]$. This follows from the fact that the operation predicate $[\![\mathsf{op}]\!]$ is a postcondition of *true* w.r.t. the operation op. The third, **(I3)** is that for any path of length m, if some *prefix* of the path $\pi[i]$ satisfies φ, and none of the variables occurring in φ are written in any operation between $\pi(i)$ and $\pi(m)$, then φ is preserved along the subsequent execution (*i.e.,* holds at all instances) and hence π satisfies φ. Formally, for every formula φ, and path π of length m, if for some $1 \leq i \leq m$, the subpath $\pi[i]$ satisfies φ, and for every $i < j \leq m$, we have $\mathsf{vars}(\varphi) \cap \mathsf{wr}(\pi(j)) = \emptyset$ then π also satisfies φ.

4 Structural Invariants

We now describe how dominators and node predicates can be combined to to efficiently compute path-sensitive invariants with which a variety of assertions can be proved efficiently. We begin by describing Dominator Invariants and then generalizing them to Structural Invariants.

4.1 Dominator Invariants

Our first result relates dominator nodes 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, if the program is in SSA form, then immediately after an operation op is executed, the system is in a state satisfying the operation predicate $[\![\mathsf{op}]\!]$. The second, is that if $\mathtt{n}'$ is a node dominating node $\mathtt{n}$, then any execution path to $\mathtt{n}$ must pass through $\mathtt{n}'$. Hence, 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 by the system. Third, we show that if $\mathtt{n}'\mathsf{D}\mathtt{n}$, then in any execution path, after the *last* occurrence of $\mathtt{n}'$, none of the variables of $\mathtt{n}'$ are ever modified. Thus, as $[\![\mathtt{n}']\!]$ held immediately after (the last occurrence of) $\mathtt{n}'$, it was preserved until execution reached $\mathtt{n}$, and hence $[\![\mathtt{n}']\!]$ is a $\mathtt{n}$-invariant. Hence, the conjunction of node predicates for all nodes dominating $\mathtt{n}$ is a $\mathtt{n}$-invariant, called the *Dominator Invariant* of $\mathtt{n}$.

Proposition 2. **[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.*

We state this as a proposition, as it follows immediately from two more general facts which we will exploit further. The first fact is that if φ is a $\mathtt{n}'$-invariant, *none of* whose variables are written in nodes *dominated by* $\mathtt{n}'$, then φ

is also a $\mathtt{n}$-invariant for all nodes $\mathtt{n}$ that are *dominated by* $\mathtt{n}'$. The second fact is that the variables of $[\![\mathtt{n}']\!]$ are not written in any node dominated by $\mathtt{n}'$, and hence $[\![\mathtt{n}']\!]$ is a $\mathtt{n}$-invariant for all nodes dominated by $\mathtt{n}'$.

Theorem 1. *For node* $\mathtt{n}$ *of a CFG in SSA form, and for all* $\mathtt{n}' \in \mathsf{D}(\mathtt{n})$,

(i) if φ *is a* $\mathtt{n}'$*-invariant such that* $\mathsf{vars}(\varphi) \cap \mathsf{wr}(\mathsf{D}^{-1}(\mathtt{n}')) = \emptyset$ *then* φ *is a* $\mathtt{n}$*-invariant, and*
(ii) the formula $[\![\mathtt{n}']\!]$ *is a* $\mathtt{n}$*-invariant.*

This theorem follows from two insights. First, in any execution path to node $\mathtt{n}$, after the last occurrence of $\mathtt{n}' \in \mathsf{D}(\mathtt{n})$, the *only nodes visited* are those that are *dominated by* $\mathtt{n}'$. This is precisely stated in (F1) from Proposition 1. To see why, suppose this was not the case. That is, suppose that *after* the last occurrence of $\mathtt{n}'$ on the path, there is some $\mathtt{n}''$ *not dominated by* $\mathtt{n}'$ that appears on the path. As along the suffix of the path from $\mathtt{n}''$ to $\mathtt{n}$ (the last node), there is no occurrence of $\mathtt{n}'$, we know there is a path π^+ from $\mathtt{n}''$ to $\mathtt{n}$ that doesn't pass through $\mathtt{n}'$. As $\mathtt{n}''$ is not dominated by $\mathtt{n}'$, there is a path π^- from the entry node to $\mathtt{n}''$ that does not visit $\mathtt{n}'$. Hence, the path $\pi^-;\ \pi^+$ is a path from the entry node to $\mathtt{n}$ that does not visit $\mathtt{n}'$, contradicting the assumption that $\mathtt{n}'$ dominates $\mathtt{n}$! This is illustrated in Figure 3(a). Hence, if φ is a $\mathtt{n}'$-invariant, none of whose variables are modifed by nodes *dominated by* $\mathtt{n}'$, then φ will always hold after execution passes through $\mathtt{n}'$ and hence φ is a $\mathtt{n}$-invariant for all nodes *dominated by* $\mathtt{n}'$ (i).

Second, the variables of $[\![\mathtt{n}']\!]$ are disjoint from those *written* in nodes dominated by $\mathtt{n}'$. This is proved by noting that SSA form ensures that that variables *written* in $\mathtt{n}'$ are trivially disjoint from those written in the dominated nodes (as each variable is assigned only in one CFG node) and that SSA form also ensures (F2) that if a node in which a variable is written is *dominated by* a node $\mathtt{n}'$ in which the variable is read, then $\mathtt{n}'$ must be a loop header (otherwise definitions dominate uses), in which case $[\![\mathtt{n}']\!]$ is *true*, *i.e.,* no variables appear in it. Combining the that the variables of $[\![\mathtt{n}']\!]$ are not written by nodes dominated by $\mathtt{n}'$ with (i) gives us the second part of Theorem 1.

4.2 Structural Invariants

As shown in Section 2, dominator invariants ignore conditional control flow merges in the code, and are often not precise enough to prove properties of interest. Therefore, we generalize dominator invariants to *structural invariants* that take nested conditionals into account to produce more precise invariants.

As shown in Section 2, the source of imprecision is $\phi-$nodes where a variable may get a value from one of two (or more) predecessors, neither of which dominates the target node. Thus, we *strengthen* the node constraints of the $\phi-$nodes using a disjunction of the recursively computed dominator invariants of the predecessors of the ϕ-nodes.

ϕ-Strengthening. The main idea we exploit is that as long as the ϕ-node is a join node (and not a loop header) we recursively compute the structural invariants of the predecessors, conjoin them with a constraint fixing the value of

the variable assigned at the ϕ-node to that which held at each predecessor, and use the *disjunction* of the resulting predicates to strengthen the node predicate of the ϕ-node. We call this process *recursive ϕ-strengthening* and illustrate it in Figure 3(b). Though the *exact* structural invariant can be computed in linear time by recursively strengthening each ϕ-node, in practice such precision is not required. Instead, we explicitly parameterize the SI with a recursion-depth bound k. Hence, 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. The first 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 second 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:

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

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 $\Psi((\mathtt{n}_e, \mathtt{n}), k)$, and the *Structural Invariant* of a node $\mathtt{n}$ is $\Psi((\mathtt{n}_e, \mathtt{n}),\ N)$.

Figure 3(b,c) 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 red) in the dominator tree, *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.

Thus, the parameter k corresponds to the "branch-width sensitivity" of the analysis w.r.t. the program structure *i.e.,* the dominator tree. This bound allows us to incrementally tune the precision of the invariant, and use coarser (and faster computed) invariants wherever possible. In addition, it gives us an easy

way to prove the soundness of the invariant generation. We have already shown in Proposition 2 that 1-SI are indeed invariants for the corresponding nodes, and we use induction on k to prove that k-SI are invariants as well.

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. $\Psi((\mathtt{n}_e, \mathtt{n}), k+1) \Rightarrow \Psi((\mathtt{n}_e, \mathtt{n}), k)$.

An immediate corollary of the above is that the Structural Invariant (i.e. the N-SI) of a node $\mathtt{n}$ is also a $\mathtt{n}$-invariant. We prove this theorem by induction on the branching depth k. We have already seen the base case (dominator invariants). To prove the inductive case, we use the induction hypothesis for $(k-1)$ to show that the disjunction of the $(k-1)$-SI computed for the predecessors of a join node $\mathtt{n}'$ is indeed a $\mathtt{n}'$-invariant. In addition, we show that the variables appearing in the $(k-1)$-SI thus computed are *never modified* in any node *dominated by* $\mathtt{n}'$. Thus, using Theorem 1 we can conclude that the strengthening $\Gamma(\mathtt{n}')$ is an $\mathtt{n}$-invariant for all nodes dominated by $\mathtt{n}'$, *i.e.,* if $\mathtt{n}'$ was a join node dominating $\mathtt{n}$, then $\Gamma(\mathtt{n}')$ is also an $\mathtt{n}$-invariant.

We now prove Theorem 2. We shall use the following facts about the variables appearing in Ψ, Γ. We say that $\mathtt{n}$ *goes to* $\mathtt{n}'$ under $\mathtt{n}''$, written $\mathtt{n} \overset{\mathtt{n}''}{\rightsquigarrow} \mathtt{n}'$ if there is a path in the CFG from $\mathtt{n}$ to $\mathtt{n}'$ such that every vertex along the path is strictly dominated by $\mathtt{n}''$.

Lemma 1. *For all nodes $\mathtt{n}_r, \mathtt{n}$ such that $\mathtt{n}_r \ \mathsf{D}\ \mathtt{n}$, and $k \in \mathbb{N}$,*

(i) $\mathsf{vars}(\Psi((\mathtt{n}_r, \mathtt{n}), k)) \subseteq \mathsf{vars}([\![\mathtt{n}_r]\!]) \cup \mathsf{vars}(\{[\![\mathtt{n}']\!] \mid \mathtt{n}' \notin \mathsf{D}^{-1}(\mathtt{n}) \wedge \mathtt{n}' \overset{\mathtt{n}_r}{\rightsquigarrow} \mathtt{n}\})$,
(ii) $\mathsf{vars}(\Gamma(\mathtt{n}, k)) \subseteq \mathsf{vars}([\![\mathsf{ID}(\mathtt{n})]\!]) \cup \mathsf{vars}(\{[\![\mathtt{n}']\!] \mid \mathtt{n}' \notin \mathsf{D}^{-1}(\mathtt{n}) \wedge \mathtt{n}' \overset{\mathsf{ID}(\mathtt{n})}{\rightsquigarrow} \mathtt{n}\})$

The proof is by simultaneous induction on k. Induction step for Ψ is straightforward, using (F1) and transitivity of D. Induction step for Γ uses that if $\mathtt{n}' \in \mathsf{pred}(\mathtt{n})$, then $\mathsf{ID}(\mathtt{n})$ dominates $\mathtt{n}'$. Using Lemma 1, we can prove the next two facts about the variables appearing in the structural constraints.

Lemma 2. *For every node $\mathtt{n}$ and $k \in \mathbb{N}$,*

(i) $\mathsf{vars}(\Gamma(\mathtt{n}, k)) \cap \mathsf{wr}(\mathsf{D}^{-1}(\mathtt{n})) = \emptyset$,
(ii) For every $\mathtt{n}' \in \mathsf{pred}(\mathtt{n})$, *if* $\mathtt{n}' \notin \mathsf{D}^{-1}(\mathtt{n})$ *then* $(\mathsf{vars}([\![\mathtt{n}']\!]) \cup \mathsf{vars}(\Psi((\mathsf{ID}(\mathtt{n}), \mathtt{n}'), k))) \cap \mathsf{wr}(\mathtt{n}) = \emptyset$.

To show (i), we show that the intersection of $\mathsf{vars}([\![\mathsf{ID}(\mathtt{n})]\!]) \cup \mathsf{vars}(\{[\![\mathtt{n}']\!] \mid \mathtt{n}' \overset{\mathsf{ID}(\mathtt{n})}{\rightsquigarrow} \mathtt{n}\})$ and $\mathsf{wr}(\mathsf{D}^{-1}(\mathtt{n}))$ is empty and then apply Lemma 1 to deduce the result. The proof is by contradiction. If this intersection is not empty, there is a $\mathtt{n}' \in \mathsf{D}^{-1}(\mathtt{n})$, and $\mathtt{n}''$ such that $\mathsf{vars}([\![\mathtt{n}'']\!]) \cap \mathsf{wr}(\mathtt{n}') \neq \emptyset$ and either (a) $\mathtt{n}'' = \mathsf{ID}(\mathtt{n})$ or (b) $\mathtt{n}'' \notin \mathsf{D}^{-1}(\mathtt{n}) \wedge \mathtt{n}'' \overset{\mathsf{ID}(\mathtt{n})}{\rightsquigarrow} \mathtt{n}$. (a) combined with (F2) implies that $\mathtt{n}''$ is a ϕ^ℓ-node, a contradiction. (b) implies that $\mathtt{n}'' \notin \mathsf{D}^{-1}(\mathtt{n})$, *i.e.,* $\mathtt{n}'' \notin \mathsf{D}^{-1}(\mathtt{n}')$, hence from (F4), as $\mathtt{n}''$ cannot be a ϕ^ℓ node, it must be a ϕ-node. From (F5), $\mathtt{n}' \in \mathsf{D}^{-1}(\mathsf{ID}(\mathtt{n}''))$, and as $\mathtt{n}' \in \mathsf{D}^{-1}(\mathsf{ID}(\mathtt{n}))$ we deduce that $\mathsf{ID}(\mathtt{n}'') = \mathsf{ID}(\mathtt{n})$. From

(F3) there is a path from $\mathtt{n}'$ to $\mathtt{n}''$ where every intermediate node is dominated by $\mathtt{n}'$ and hence by $\mathtt{n}$. From (b) there is a path from $\mathtt{n}''$ to $\mathtt{n}$ where every node is strictly dominated by $\mathsf{ID}(\mathtt{n})$. Hence there is a cycle with $\mathtt{n}''$ and $\mathtt{n}$. By (F6) there must be a single node in the cycle that dominates all the others. The only node in this cycle that dominates $\mathtt{n}''$ is itself. Yet $\mathtt{n}''$ doesnt dominate $\mathtt{n}$ and hence we have a contradiction.

The proof of (ii) is very much like (i). Only now we have a path from the ϕ node ($\mathtt{n}''$ above) to the predecessor $\mathtt{n}'$, and a path from $\mathtt{n}$ to the write node and then to the ϕ-node ($\mathtt{n}''$ above), where all intermediate nodes are dominated by $\mathtt{n}$. The edge from the predecessor $\mathtt{n}'$ to $\mathtt{n}$ completes the cycle.

To prove Theorem 2, we shall show simultaneously by induction on k that for CFGs in SSA-form, that for all $k \geq 0$:
(S1) for any $\mathtt{n}'$, $\Gamma(\mathtt{n}', k)$ is a $\mathtt{n}'$-invariant, and
(S2) for any nodes $\mathtt{n}, \mathtt{n}_r$ with $\mathtt{n}_r \in \mathsf{D}(\mathtt{n})$, $\Psi((\mathtt{n}_r, \mathtt{n}), k)$ is a $\mathtt{n}$-invariant.
Base sase: ($k = 0$). As $\Gamma(\mathtt{n}', 0) = \mathit{true}$ and $\Psi((\mathtt{n}_r, \mathtt{n}), 0) = \mathit{true}$, both (S1) and (S2) trivially holds.
Induction Step: We assume by the induction hypothesis that (S1), (S2) hold for all $k \leq d$. We shall now prove that (S1) and (S2) hold for $k = d + 1$, and hence the theorem.

To show (S1) for $k = d+1$ we shall show that every path ending at $\mathtt{n}'$ satisfies $\Gamma(\mathtt{n}', d + 1)$. This trivially holds if $\mathsf{pred}(\mathtt{n}') \cap \mathsf{D}^{-1}(\mathtt{n}') \neq \emptyset$, so let us assume that the intersection is empty. Consider any such path π of length m. Let $\mathtt{n} = \pi(m-1)$, be the penultimate node along the path. (S2) implies that $\varphi \equiv \Psi((\mathsf{ID}(\mathtt{n}'), \mathtt{n}), d)$ is a $\mathtt{n}$-invariant, and so $\pi[m-1]$ satisfies φ. By (I2) $\pi[m-1]$ satisfies $[\![\mathtt{n}]\!]$. As $\mathsf{pred}(\mathtt{n}') \cap \mathsf{D}^{-1}(\mathtt{n}') = \emptyset$ by Lemma 2(ii) we conclude $\mathsf{vars}(\varphi \wedge [\![\mathtt{n}]\!]) \cap \mathsf{wr}(\mathtt{n}') = \emptyset$ and so by (I3), π satisfies φ. Finally, if $\mathtt{n}'$ is a ϕ node, and $\mathtt{n}$ the kth predecessor, then the live value at $\mathtt{n}'$ is the one assigned at node $\mathtt{n}$, and so π satisfies the constraints $\mathtt{l} = \mathtt{x}_k$. Hence, by (I1) π satisfies the conjunction $[\![\mathtt{n}]\!] \wedge \Phi_{\mathtt{n}}(\mathtt{n}') \wedge \Psi((\mathsf{ID}(\mathtt{n}'), \mathtt{n}), d)$, and so, the weaker formula $\Gamma(\mathtt{n}', d + 1)$. Thus (S1) holds for $k = d + 1$.

To prove (S2) for $k = d + 1$, we use fact (I1) to deduce that it suffices to show that for each $\mathtt{n}'$ in $\mathsf{D}(\mathtt{n})$, the formulas $[\![\mathtt{n}']\!]$ and $\Gamma(\mathtt{n}', d + 1)$ are $\mathtt{n}$-invariants. Theorem 1(ii) implies the first condition, *i.e.,* that $[\![\mathtt{n}']\!]$ is a $\mathtt{n}$-invariant. Further, (S1) with $k = d+1$ implies that $\Gamma(\mathtt{n}', d + 1)$ is a $\mathtt{n}'$-invariant. Lemma 2(i) together with Theorem 1(i) implies that $\Gamma(\mathtt{n}', d + 1)$ is a $\mathtt{n}$-invariant as well.

Incorporating Other Invariants. Suppose $\mathcal{I}$ is a function that maps each node $\mathtt{n}$ of the control flow graph to a predicate $\mathcal{I}(\mathtt{n})$ that is a $\mathtt{n}$-invariant. For a predicate φ and a set V of variables, let $\exists V.\varphi$ be the predicate obtained by existentially quantifying all variables in V from φ. We generalize the definition of the function Ψ to $\Psi_{\mathcal{I}}$ by incorporating this invariant information as follows:

$$\Psi_{\mathcal{I}}((\mathtt{n}_r, \mathtt{n}), k) \equiv [\![\mathtt{n}]\!] \wedge \exists \mathsf{wr}(\mathsf{D}^{-1}(\mathtt{n})).\mathcal{I}(\mathtt{n}) \wedge \bigwedge_{\mathtt{n}' \in \mathsf{D}(\mathtt{n}) \cap \mathsf{D}^{-1}\mathtt{n}_r} \Gamma(\mathtt{n}', k)$$

if $k > 0$ and $\mathtt{n}_r \neq \mathtt{n}$ and $\exists \mathsf{wr}(\mathsf{D}^{-1}(\mathtt{n})).\mathcal{I}(\mathtt{n})$ otherwise; and use $\Psi_{\mathcal{I}}$ instead of Ψ in the definition of Γ. This generalization preserves the properties of (k-)structural

invariants, and is useful in practice to avoid certain false positives arising out of loops.

Example 7. Consider an array bounds check:

```
for(i=0;i<a_len;i++){ assert(0<=i <a_len); ..a[i].. }
```

For any k, the assertion is not provable using k-SIs. However, given the invariant $0 \leq i$, obtained *e.g.* from a signs abstract interpreter, the generalized 1-SI can prove the assertion.

5 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 there are no global variables. We extend k-structural invariants to the interprocedural case. We describe two approaches: summarization and abstract summarization.

5.1 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 $[\![\mathtt{op}]\!]$ to the new operations as follows. 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 $G_f = (N^f, E^f, \mathtt{n}_e^f, \mathtt{n}_x^f)$. Let L' be a set of fresh names for each local variable in L. We define $[\![\mathtt{l} := f(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]$ as

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

and $[\![\mathtt{return}(\mathtt{ret})]\!] = \mathit{true}$. Intuitively, we recursively construct the k-SI for the exit node of f and substitute the formal parameters and return variable in the expression. This k-SI is the summary of f. Finally, to avoid name clashes for multiple calls to f, we substitute all local variables of f in the expression with fresh names L' (*i.e.,* existentially quantify all locals). The constraint from a return operation is *true*. 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.

5.2 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 $(p, p') \in S$, there exists an execution of the function starting from a state satisfying p to a state satisfying p'.

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 $\mathtt{n}_x$ of f with respect to the entry point 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 equivalent to *true*, abstract summarization can lead to unsoundness. To regain soundness, we add additional assertions to the program. At each call site $\mathtt{x} := f(e_1, \ldots, e_n)$, we add the assertion $\mathtt{assert}(\bigvee P)[e_1/\mathtt{x}_1, \ldots, e_n/\mathtt{x}_n][L'/L]$ This 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 $(\mathit{true}, \mathit{true})$ to check 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. While it requires user-supplied predicates, for many properties, including those studied in our experiments (see also [24]), we can come up with automatic heuristics to generate predicates.

6 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, psichecks if the k-structural invariant implies the assertion using the Simplify theorem prover [10]. The implementation is staged in five parts: alias analysis, SSA conversion, dominator tree construction, construction of the k-structural invariant, 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. After alias analysis, structures are broken up into their individual fields. We do the structure expansion *after* aliasing for scalability of our alias analysis. We expand fields so that each field can have an individual SSA numbering. In other words when one field is written, only the field should change its number, not the whole structure. We cache structural invariants to avoid recomputation. Further, we compute a backward slice with respect to the variables in the assertion and only keep constraints that may transitively influence these variables.

We ran three sets of experiments with our tool: to check 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.

6.1 Tagged Unions

Tagged unions are checked by adding an assertion describing the predicate that must hold when a certain field is accessed or cast is made before that access or cast is made. We added these assertions manually. Table 1 summarizes our results with $k = 2$.

ICMP is a protocol for error notification on the internet. It has nine different structures determined by its type and each type is further constrained with an 8 bit code. Our tool examined the ICMP implementation in the FreeBSD kernel. It found three false positives with $k = 1$. However, adding path sensitivity with $k = 2$ reduced the false positives to 1. GTK+ Drawing Kit (GDK) [16] is a set of lower level APIs that are used in the Gimp Tool Kit (GTK+). The GDK implementation uses a structure that represents a container for all events. This structure contains a union of 20 kinds of other structures describing events, and a field representing the type of event being contained. psifound 3 false positives with $k = 1$, which was reduced to 1 with $k = 2$. Lua [18] is a dynamically typed language used for extensible application development. The

Lua interpreter contains a single structure that can contain different values (tables, strings, numbers, etc.) and a field that represents which type is currently contained in the union.

Program	LOC	Safe	FP	Time (s)
`icmp`	7K	7	1	1
`gdk`	16K	7	1	1
`lua`	18K	41	12	682

Table 1. Tagged union checking

We checked 69 assertions and found 14 false positives. 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. Other false positives are due to modeling pointer arithmetic and data structures. Four false positives were due to fields accessed using pointer arithmetic, and one false positive was due to modifying a field through an union and accessing that field by another name (we do not currently model memory layout precisely).

program	LOC	functions	asserts	total	ok	error	pointers	List	loops	unclassified	time(s)	cqual
`scc`	16,341	638	36	57	47	2	7	0	0	1	38	60
`DAC960`	23,740	763	46	54	38	0	10	0	4	2	141	N/A
`af_netrom`	22,414	958	23	25	21	0	0	3	1	3	12	20
`af_rose`	22,606	958	15	29	28	0	0	0	0	1	7	9
`as-iosched`	14,165	576	10	17	10	0	4	0	0	3	8	4
`elevator`	12,078	512	2	3	3	0	0	0	0	0	1	0
`floppy`	17,993	696	30	48	43	0	0	0	2	3	35	48
`genhd`	12,418	529	4	6	6	0	0	0	0	0	2	0
`ll_rw_blk`	15,020	625	8	30	25	0	0	0	2	3	8	N/A
`nr_route`	18,438	788	19	34	30	0	0	1	1	2	9	20
`wavelan_cs`	17,491	621	19	35	30	1	4	0	0	0	14	4
`rose_route`	42,314	953	51	73	55	13	0	0	3	2	35	31
Totals	235,018	8,617	263	414	336	16	25	4	13	20	310	196

Table 2. Lock experiments. Asserts gives the original number of asserts, and total gives the total number of asserts after pre- and post-conditions are added. ok gives the number of asserts proved safe. error the number of bugs. False positives are broken into pointers, lists, loops, and unclassified errors. Cqual shows the false positives from Cqual (N/A indicates we did not run Cqual).

6.2 Locking

The second set of experiments checked double locking errors in the Linux kernel. Double locking has been extensively studied using dataflow analysis [15] and symbolic execution [24]. Double locking occurs when locking something that already has been locked or unlocking something that already has been unlocked. Locking a lock twice in a row results in a deadlock while unlocking a lock twice is considered bad programming style and, depending on kernel implementation, 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]. However, instead of user provided predicates, we use a simple heuristic to guess predicates that works well for most functions. 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. After all preconditions, postconditions, assertions and summarization instructions have been added, our structural invariant algorithm is run with a path depth $k = 2$. We found increasing $k > 2$ does not reduce the number of false positives in our experiments.

Table 2 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. The unsound treatment of multiple files missed one bug (caught by Saturn). The bug required function summaries from another source file, and would be caught if we collected all summarized functions. On the other hand, we found errors in wavelan and rose_route not mentioned in the Saturn bug database. The false positives are in three categories: pointers in summaries (25), getting locks from dynamic data structures or external functions (4), and loops (13). There are 20 additional errors we have not classified yet.

Pointers in summaries 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 is not a global variable. In all our testcases, this is due to code like:

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

This can be solved by changing the preconditions and postconditions 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.

Figure 4 illustrates a loop false positive. This is essentially the example from Section 2, where a lock is acquired and released in a loop. Interestingly, this example can be proved using Cqual, showing the orthogonality of the two methods. Other false positives relate to dynamic data structures (where locks are stored in lists) or pointers returned from external functions.

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 terms of false positives, we lag slightly behind, because we soundly handle loops and do not give up on any functions analyzed. In comparison with Cqual [15], our approach takes longer to analyze, but has fewer false positives. Running Cqual on 10 of our 12 device driver examples resulted in 196 type errors, even though Cqual does handle the loop false positives. Further, in contrast to Cqual, we get at most one message per assertion site, making it easier to track down false errors.

6.3 Privilege Levels

Finally, we checked whether a Unix setuid program gives up its owner privileges before executing certain system calls [5]. In unix systems, programs have privileges associated with users. The user associated with the program's privileges is the effective user. Normally, a program will execute under the permission of the user executing the program. However, *suid programs* run with the permissions of the owner (usually root). These programs have root privileges when they are started, which are required to access certain system resources that only root can access. 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. While this simple model does not take account of all the complex nuances of `setuid` and `seteuid` behavior [6], it soundly models the property that every system call must be made with the privilege of a non-root-user.

We examine three programs: OpenSSH (the widely used secure shell program), GNU Privacy Guard (open source pgp), and mtr (a network diagnostic tool). We used OpenSSH 2.9.9p which is an suid process. 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 predicates. This caused a significant number of extra assertions to be added (16 out of 270

assertions total were from the property), however, the summarization made our technique scale well. Further, $k = 1$ was enough to prove all assertions and there were no false positives.

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

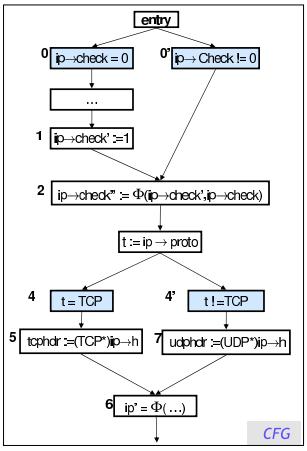
Our results are shown in Table 3, where the time does not include time for alias analysis. Experiments ran in the order of minutes. However, as the size of the programs increased, CIL's alias analysis became the bottleneck. It took 610s for OpenSSH and did not terminate for gpg within 6 hours. However, since the id bit is not aliased to any program variable and is only assigned constants, and we additionally check that the k-SI is satisfiable, we can prove in this case that our technique is sound without the alias analysis.

References

1. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.
2. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, LNCS 1579, pages 193–207. Springer, 1999.
3. 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*, pages 196–207, 2003.
4. S. Chaki, J. Ouaknine, K. Yorav, and E.M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *SoftMC 03*, 2003.
5. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of c code. In *NDSS*, pages 171–185, 2004.
6. H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.
7. E.M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, LNCS 2988, pages 168–176. Springer, 2004.
8. 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 TOPLAS*, 13:451–490, 1991.
9. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68. ACM, 2002.
10. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

11. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*. Usenix, 2000.
12. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245. ACM, 2002.
13. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205. ACM, 2000.
14. R.W. Floyd. Assigning meanings to programs. In *Math. Aspects of Comp. Sci.*, pages 19–32. American Mathematical Society, 1967.
15. J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12. ACM, 2002.
16. The gimp toolkit. http://www.gtk.org/.
17. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
18. R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of lua 5.0. In *Proc. of IX Brazilian Symp. on Prog. Lang.*, pages 63–75. ACM, 2005.
19. D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the alloy constraint analyzer. In *ICSE*, pages 730–733, 2000.
20. T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flow-graph. *ACM TOPLAS*, 1(1):121–141, 1979.
21. M. Musuvathi, D.Y.W. Park, A. Chou, D.R. Engler, and D.L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI*. 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*, LNCS 2304, pages 213–228. Springer, 2002.
23. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
24. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL*, pages 351–363. ACM, 2005.

```
struct iphdr {
  int check;
  int proto; // TCP or UDP
  ...
  char *h; // TCP or UDP payload
};
examle1(struct iphdr *ip) {
1:if (!ip->check) {
    ... // perform CRC 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;
    ...
  }
}
```



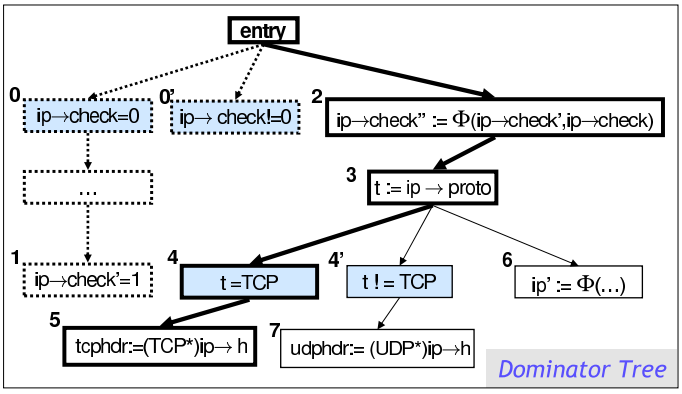


Fig. 1. (a) Example 1 (b) CFG in SSA form (c) Dominator Tree

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

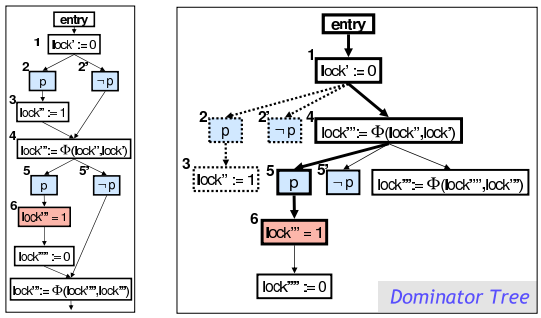


Fig. 2. (a) Example 2 (b) CFG (c) Dominator Tree

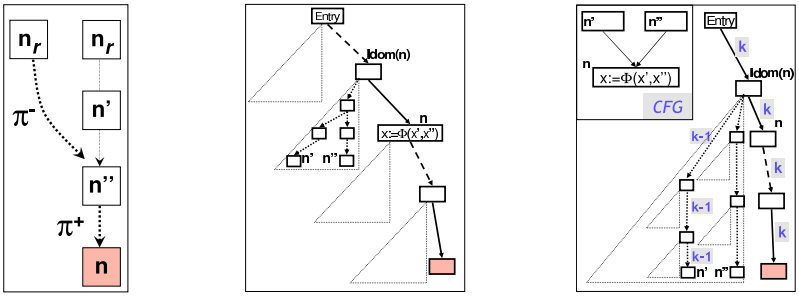


Fig. 3. (a) If a node $\mathtt{n}''$ not dominated by $\mathtt{n}'$ appears after the last occurrence of $\mathtt{n}'$ on a path to $\mathtt{n}$, then $\mathtt{n}$ is not dominated by $\mathtt{n}'$. (b) ϕ-strengthening: For a join ϕ-node $\mathtt{n}$, the strengthening $\Gamma(\mathtt{n})$ is the disjunction of the SIs of the two predecessors $\mathtt{n}', \mathtt{n}''$ of $\mathtt{n}$, which are in the tree "hanging off" the $\mathsf{ID}(\mathtt{n})$ (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 $\mathsf{ID}(\mathtt{n})$, recursively exploring the trees hanging off the inner paths.

```
spin_lock_irqsave(&Controller->queue_lock, flags);
. . .
while ( ... ){
    spin_unlock_irq(&Controller->queue_lock);
    __wait_event(...);
    spin_lock_irq(&Controller->queue_lock);
}
. . .
spin_unlock_irqrestore(&Controller->queue_lock,flags);
```

Fig. 4. Loop false positive in DAC960.c