Loop Summarization using Abstract Transformers*

Daniel Kroening[1], Natasha Sharygina[2,5], Stefano Tonetta[3], Aliaksei Tsitovich[2], and Christoph M. Wintersteiger[4]

[1] Oxford University, Computing Laboratory, UK
[2] University of Lugano, Switzerland
[3] Fondazione Bruno Kessler, Trento, Italy
[4] Computer Systems Institute, ETH Zurich, Switzerland
[5] School of Computer Science, Carnegie Mellon University, USA

Abstract. Existing program analysis tools that implement abstraction rely on saturating procedures to compute over-approximations of fixpoints. As an alternative, we propose a new algorithm to compute an over-approximation of the set of reachable states of a program by replacing loops in the control flow graph by their abstract transformer. Our technique is able to generate diagnostic information in case of property violations, which we call *leaping counterexamples*. We have implemented this technique and report experimental results on a set of large ANSI-C programs using abstract domains that focus on properties related to string-buffers.

1 Introduction

Abstract Interpretation [1] is a framework for the approximative analysis of discrete transition systems, and is based on fixpoint computations. It is frequently applied to verify reachability properties of software programs. Abstract interpretation is performed with respect to an *abstract domain*, which is an approximate representation of sets of concrete values. Instances are numerical abstract domains such as intervals [1] and polyhedra [2], or specialized domains as, for example, a domain specific for heap-manipulating programs. In abstract interpretation, the behavior of a program is evaluated over the abstract domain using an *abstract transformer*. This is iterated until the set of abstract states saturates, i.e., an abstract fixpoint is reached. If certain formal constraints between the abstract and concrete domains are met, this abstract fixpoint is guaranteed to be an over-approximation of the set of reachable states of the original program.

A main issue in abstract interpretation is the number of iterations required to reach the abstract fixpoint. On large benchmarks, a thousand iterations is commonly observed, even when using simplistic abstract domains. Thus, many tools implementing abstract interpretation apply *widening* in order to accelerate

* Supported by the Swiss National Science Foundation grant no. 200021-111687 and an award from IBM research.

convergence. Widening, however, may yield imprecision, and thus, the abstract fixpoint may not be strong enough to prove the desired property [3].

We propose a novel technique to address this problem, which uses a *symbolic abstract transformer* [4]. A symbolic abstract transformer for a given program fragment is a relation over a pair of abstract states $\hat{s}, \hat{s}'$ that holds if the fragment transforms $\hat{s}$ into $\hat{s}'$. We propose to apply the transformer to perform sound *summarization*, i.e., to replace parts of the program by a smaller representative. In particular, we use the transformer to summarize loops and (recursion-free) function calls.

The symbolic abstract transformer is usually computed by checking if a given abstract transition is consistent with the semantics of a statement [4, 5]. Our technique generalizes the abstract transformer computation and applies it to program fragments: given an abstract transition relation, we check if it is consistent with the program semantics. This way, we can tailor the abstraction to each program fragment. In particular, for loop-free programs, we precisely encode their semantics into symbolic formulas. For loops, we exploit the symbolic transformer of the loop body to infer invariants of the loop. This is implemented by means of a sequence of calls to a decision procedure for the program logic.

When applied starting from the inner-most loops and the leaves of the call graph, the run-time of the resulting procedure becomes linear in the number of looping constructs in the program, and thus, is often much smaller than the number of iterations required by the traditional saturation procedure. We show soundness of the procedure and discuss its precision compared to the conventional approach on a given abstract domain. In case the property fails, a diagnostic counterexample can be obtained, which we call *leaping counterexample*. This diagnostic information is often very helpful for understanding the nature of the problem, and is considered a major plus for program analysis tools. Additionally, our technique *localizes* the abstract domains: we use different abstract domains for different parts of the code. This further improves the scalability of the analysis.

We implemented the procedure in a tool called LoopFrog and applied it to search for buffer-overflow errors in well-known UNIX programs. Our experimental results demonstrate that the procedure is more precise than any other tool we compared with. Moreover, it scales to large programs even if complex abstract domains are used. In summary, the contributions of this paper are:

- We introduce a new technique for program abstraction by using symbolic abstract transformers for summarization of loops and function calls. The technique is sound and has the advantage that the run-time is linear in the number of looping constructs.
- In contrast to most other implementations of abstract interpretation, our analysis technique produces counterexamples which can be used to diagnose the property violation. Moreover, the counterexamples can be used to refine the abstract domains.

Related work. Other work on analysis using summaries of functions is quite extensive (see a nice survey in [6]) and dates back to Cousot and Halbwachs [2],

and Sharir and Pnueli [7]. In a lot of projects, function summaries are created for alias analysis or points-to analysis, or are intended for the analysis of program fragments. As a result, these algorithms are either specialized to particular problems and deal with fairly simple abstract domains or are restricted to analysis of parts of the program. An instance is the summarization of library functions in [6]. In contrast, our technique aims at computing a summary for the entire program, and is applicable to complex abstract domains. The principal novelty of our technique is that it is a general-purpose loop summarization method that (unlike many other tools) is not limited to special classes of faults.

Similarly to our technique, the Saturn tool [8] computes a summary of a function with respect to an abstract domain using a SAT-based approach to improve scalability. However, in favor of scalability, Saturn simply unwinds loops a constant number of times, and thus, bugs that require more iterations are missed. Similarly to Saturn, the Spear tool [9] summarizes the effect of larger functions, which improves the scalability of the tool dramatically. However, as in the case of Saturn, loops are unwound only once.

SAT-solvers, SAT-based decision procedures, and constraint solvers are frequently applied in program verification. Instances are the tools Alloy [10] and CBMC [11]. The SAT-based approach is also suitable for computing abstractions, as, for example, in [8, 5, 4] (see detailed discussion in Sec. 2.3). The technique reported in this paper also uses the flexibility of a SAT-based decision procedure for a combination of theories to compute loop summaries.

One of the benefits of our approach is its ability to generate diagnostic information for failed properties. This is usually considered a distinguishing feature of *model checking* [12], and is rarely found in tools based on abstract interpretation. Counterexamples aid the diagnosis of errors, and may also be used to filter spurious warnings.

2 Background

2.1 Notation

In this section we introduce the basic concepts of abstract interpretation [1, 13]. Let U denote the universe where the values of the program variables are drawn from. The set L of *elementary commands* consists of tests L_T and assignments L_A, i.e., $L = L_T \dot{\cup} L_A$, where a test $q \in L_T$ is a predicate over $dom(q) \subseteq U$. An assignment $e \in L_A$ is a total map from $dom(e) \subseteq U$ to U. Given $q \in L_T$, we denote with $\overline{q}$ the predicate over $dom(q)$ such that $q(u) = \neg\overline{q}(u)$ for all $u \in dom(q)$.

A program π is formalized as the pair $\langle U, G\rangle$, where U is the universe and G is a *program graph* [13]. A program graph is a tuple $\langle V, E, v_i, v_o, C\rangle$, where

- V is a finite non-empty set of vertices called *program locations*.
- $v_i \in V$ is the initial location.
- $v_o \in V$ is the final location.

```
p=a;
while(*p!=0){
 if(*p=='/')
  *p=0;
 p++;
}
```

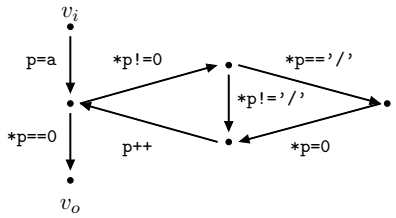


Fig. 1. The running example

- $E \subseteq V \times V$ is a non-empty set of edges; E^* denotes the set of *paths*, i.e., the set of finite sequences of edges.
- $C : E \to L$ associates a command with each edge.

We write L^* for the set of sequences of commands. Given a program π, the set $paths(\pi) \subseteq L^*$ contains the sequence $C(e_1), \ldots, C(e_n)$ for every $\langle e_1, .., e_n \rangle \in E^*$.

Example 1. We use the program fragment in Figure 1 as running example. On the left-hand side, we provide the C version. On the right-hand side, we depict its program graph.

The (concrete) semantics of a program is given by the pair $\langle A, \tau \rangle$, where

- A is the set of *assertions* of the program, where each assertion $P \in A$ is a predicate over U; $A(\Rightarrow, false, true, \vee, \wedge)$ is a complete Boolean lattice;
- $\tau : L \to (A \to A)$ is the (concrete) predicate transformer.

In forward semantic analysis, τ represents the strongest post-condition. The analysis of a program determines which assertions are true in each program location by simulating the program from the initial location to that particular program location.

2.2 Abstract Interpretation

An *abstract interpretation* is a pair $\langle \hat{A}, t \rangle$, where $\hat{A}$ is a complete lattice $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$, and $t : L \to (\hat{A} \to \hat{A})$ is a predicate transformer. Note that $\langle A, \tau \rangle$ is a particular abstract interpretation called the *concrete interpretation*. In the following, we assume that for every command $c \in L$, the function $t(c)$ is monotone (which is the case for all natural predicate transformers). Given a predicate transformer t, the function $\tilde{t} : L^* \to (\hat{A} \to \hat{A})$ is recursively defined as follows:

$$\tilde{t}(p)(\phi) = \begin{cases} \phi & \text{if } p \text{ is empty} \\ \tilde{t}(e)(t(q)(\phi)) & \text{if } p = q; e \text{ for some } q \in L,\ e \in L^*. \end{cases}$$

Example 2. We continue the running example (Fig. 1). Consider an abstract domain where abstract states are a four-tuple $\langle p_a, z_a, s_a, l_a \rangle$. The first member, p_a is the offset of the pointer p from the base address of the array a (i.e. $p - a$ in our example), the Boolean z_a holds if a contains the zero character, the Boolean s_a holds if a contains

the slash character, l_a is the index of the first zero character if present. The predicate transformer t is defined as follows:
$t(p = a)(\phi) = \phi[p_a := 0]$ for any assertion ϕ;
$t(*p\,!= 0)(\phi) = \phi \wedge (p_a \neq l_a)$ for any assertion ϕ;
$t(*p == 0)(\phi) = \phi \wedge z_a \wedge (p_a \geq l_a)$ for any assertion ϕ;
$t(*p ==' /')(\phi) = \phi \wedge s_a$ for any assertion ϕ;
$t(*p\,!=' /')(\phi) = \phi$ for any assertion ϕ;

$$t(*p = 0)(\phi) = \begin{cases} \bot & \text{if } \phi \Rightarrow \bot; \\ \phi[z_a := true, l_a := p_a] & \text{if } \phi \Rightarrow (p_a < l_a), \phi \neq \bot; \\ \phi[z_a := true] & \text{otherwise} \end{cases}$$

$t(p{+}{+})(\phi) = \phi[p_a := p_a + 1]$ for any assertion ϕ.
(We used $\phi[x := v]$ to denote an assertion equal to ϕ apart from the variable x that takes value v.)

Given a program π, an abstract interpretation $\langle \hat{A}, t \rangle$, and an element $\phi \in \hat{A}$, we define the *Merge Over all Paths* $MOP_\pi(t, \phi)$ as the element $\bigsqcup_{p \in paths(\pi)} \tilde{t}(p)(\phi)$.

Given two complete lattices $\hat{A}(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ and $\hat{A}'(\sqsubseteq', \bot', \top', \sqcup', \sqcap')$, the pair of functions $\langle \alpha, \gamma \rangle$, with $\alpha : \hat{A} \to \hat{A}'$ and $\gamma : \hat{A}' \to \hat{A}$ is a *Galois connection* iff α and γ are monotone and 1) for all $\phi \in \hat{A}$, $\phi \sqsubseteq \gamma(\alpha(\phi))$, and 2) for all $\phi' \in \hat{A}'$, $\alpha(\gamma(\phi')) \sqsubseteq' \phi'$.

An abstract interpretation $\langle \hat{A}, t \rangle$ is a *correct over-approximation* of the concrete interpretation $\langle A, \tau \rangle$ iff there exists a Galois connection $\langle \alpha, \gamma \rangle$ such that for all $\phi \in \hat{A}$ and $P \in A$, if $P \Rightarrow \gamma(\phi)$, then $\alpha(MOP_\pi(\tau, P)) \sqsubseteq MOP_\pi(t, \phi)$ (i.e., $MOP_\pi(\tau, P) \Rightarrow \gamma(MOP_\pi(t, \phi))$).

2.3 A SAT-based Abstract Transformer

In order to implement abstract interpretation for a given abstract domain, an algorithmic description of the abstract predicate transformer $t(p)$ for a specific command $p \in L$ is required. Reps et al. describe an algorithm that implements the *best possible* (i.e., most precise) abstract transformer for a given finite-height abstract domain [4]. Graf and Saïdi's algorithm for constructing predicate abstractions [14] is identified as a special case.

The algorithm has two inputs: a formula $F_{\tau(q)}$, which represents a command $q \in L$ symbolically, and an assertion $\phi \in \hat{A}$. It returns the image of the predicate transformer $t(q)(\phi)$. The formula $F_{\tau(q)}$ is passed to a decision procedure, which is expected to provide a satisfying assignment to the variables. The assignment represents one concrete transition $P, P' \in A$. The transition is abstracted into a pair $\phi, \phi' \in \hat{A}$, and a blocking constraint is added to remove this satisfying assignment. The algorithm iterates until the formula becomes unsatisfiable. An instance of the algorithm for the case of predicate abstraction is the implementation of SatAbs described in [5]. SatAbs uses a propositional SAT-solver as decision procedure for bit-vector arithmetic. The procedure is worst-case exponential in the number of predicates, and thus, alternatives have been explored. In [15, 16] a symbolic decision procedure generates a symbolic formula that represents the set of all solutions. In [17], a first-order formula is used and the computation of all solutions is carried out by a SAT modulo theories (SMT)

solver. In [18], a similar technique is proposed where BDDs are used in order to efficiently deal with the Boolean component of $F_{\tau(q)}$.

3 Summarization using Symbolic Abstract Transformers

3.1 Abstract Summarization

The idea of summarization is to replace a code fragment, e.g., a procedure of the program, by a *summary*, which is a (smaller) representation of the behavior of the fragment. Computing an exact summary of a program (fragment) is in general undecidable. We therefore settle for an over-approximation. We formalize the conditions the summary must fulfill in order to have a semantics that over-approximates the original program.

We extend the definition of a correct over-approximation (see Sec. 2) to programs. Given two programs π and π' on the same universe U, we say that π' is a *correct over-approximation* of π iff for all $P \in A(\Rightarrow, false, true, \vee, \wedge)$, $MOP_{\pi}(\tau, P) \Rightarrow MOP_{\pi'}(\tau, P)$.

Definition 1 (Abstract Summary). *Given a program π, and an abstract interpretation $\langle \hat{A}, t \rangle$ with a Galois connection $\langle \alpha, \gamma \rangle$ with $\langle A, \tau \rangle$, we denote the* abstract summary *of π by $Sum_{\langle \hat{A},t \rangle}(\pi)$. It is defined as the program $\langle U, G \rangle$, where $G = \langle \{v_i, v_o\}, \{\langle v_i, v_o \rangle\}, v_i, v_o, C \rangle$ and $C(\langle v_i, v_o \rangle)$ is a new (concrete) command a such that $\tau(a)(P) = \gamma(MOP_{\pi}(t, \alpha(P)))$.*

Lemma 1. *If $\langle \hat{A}, t \rangle$ is a correct over-approximation of $\langle A, \tau \rangle$, the abstract summary $Sum_{\langle \hat{A},t \rangle}(\pi)$ is a correct over-approximation of π.*

We now discuss algorithms for computing abstract summaries. Our summarization technique is first applied to particular fragments of the program, specifically to loop-free and single-loop programs. In Section 3.4, we use these procedures as subroutines to obtain the summarization of an arbitrary program. We formalize code fragments as *program sub-graphs*.

Definition 2. *Given two program graphs $G = \langle V, E, v_i, v_o, C \rangle$ and $G' = \langle V', E', v'_i, v'_o, C' \rangle$, G' is a* program sub-graph *of G iff $V' \subseteq V$, $E' \subseteq E$, and $C'(e) = C(e)$ for every edge $e \in E'$.*

3.2 Summarization of Loop-Free Programs

Obtaining $MOP_{\pi}(t, \phi)$ is as hard as assertion checking on the original program. Nevertheless, there are restricted cases where it is possible to represent $MOP_{\pi}(t, \phi)$ using a symbolic predicate transformer.

Let us consider a program π with a finite number of paths, in particular, a program whose program graph does not contain any cycle. A program graph $G = \langle V, E, v_i, v_o, C \rangle$ is *loop free* iff G is a directed acyclic graph.

In the case of a loop-free program π, we can compute a precise (not abstract) summary by means of a formula F_{π} that represents the concrete behavior of π.

This formula is obtained by converting π to a static single assignment (SSA) form, whose size is linear in the size of π. The details of this step are beyond the scope of this paper; see [11].

Example 3. We continue the running example (Fig. 1). The symbolic transformer of the loop body π' is represented by:
$((*p =' /' \wedge a' = a[*p = 0]) \vee (*p \neq' /' \wedge a' = a)) \wedge (p' = p + 1)$.

Recall the abstract domain from Ex. 2. We can deduce that:

1. if $m < n$, then $MOP_{\pi'}(t, (p_a = m \wedge z_a \wedge (l_a = n) \wedge \neg s_a)) = (p_a = m + 1 \wedge z_a \wedge l_a = n \wedge \neg s_a)$
2. $MOP_{\pi'}(t, z_a) = z_a$.

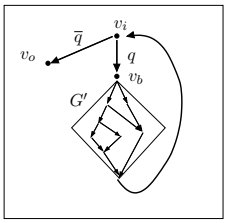
This example highlights the generic nature of our technique. For instance, case 1 of the example cannot be obtained by means of predicate abstraction because it requires an infinite number of predicates. Also, the algorithm presented in [4] cannot handle this example because assuming the string length has no a-priori bound, the lattice of the abstract interpretation has infinite height.

3.3 Summarization of Single-Loop Programs

We now consider a program that consists of a single loop.

Definition 3. *A program $\pi = \langle U, G\rangle$ is a* single-loop program *iff $G = \langle V, E, v_i, v_o, C\rangle$ and there exists a program sub-graph G' and a test $q \in L_T$ such that*

- *$G' = \langle V', E', v_b, v_i, C'\rangle$ with*
  - *$V' = V \setminus \{v_o\}$,*
  - *$E' = E \setminus \{\langle v_i, v_o\rangle, \langle v_i, v_b\rangle\}$,*
  - *$C'(e) = C(e)$ for all $e \in E'$,*
  - *G' is loop free.*
- *$C(\langle v_i, v_b\rangle) = q$, $C(\langle v_i, v_o\rangle) = \overline{q}$.*



The following can be seen as the "abstract interpretation analog" of Hoare's rule for `while` loops.

Theorem 1. *Given a single-loop program π with guard q and loop body π', and an abstract interpretation $\langle \hat{A}, t\rangle$, let ψ be an assertion satisfying $MOP_{\pi'}(t, t(q)(\psi)) \sqsubseteq \psi$ and let $\langle \hat{A}, t_\psi\rangle$ be a new abstract interpretation s.t.*

$$MOP_\pi(t_\psi, \phi) = \begin{cases} t(\overline{q})(\psi) & \text{if } \phi \sqsubseteq \psi \\ \top & \text{elsewhere.} \end{cases}$$

If $\langle \hat{A}, t\rangle$ is a correct over-approximation, then $\langle \hat{A}, t_\psi\rangle$ is a correct over-approximation as well.

In other words, if we apply the predicate transformer of the test q and then the transformer of the loop body π' to the assertion ψ, and we obtain an assertion at least as strong as ψ, then ψ is an invariant of the loop. If a stronger assertion ϕ holds before the loop, the predicate transformer of $\overline{q}$ applied to ϕ holds afterwards.

Theorem 1 gives rise to a summarization algorithm. Given a program fragment and an abstract domain, we heuristically provide a set of formulas that encode that a (possibly infinite) set of assertions ψ are invariant (for example, $x' = x$ encodes that every ψ defined as $x = c$, with c a value in the domain U, is an invariant); we apply a decision procedure to check if the formulas are satisfiable. The construction of the summary is then straightforward: given a single-loop program π, an abstract interpretation $\langle \hat{A}, t \rangle$, and an invariant ψ for the loop body, let $\langle \hat{A}, t_\psi \rangle$ be the abstract interpretation as defined in Theorem 1. We denote the summary $Sum_{\langle \hat{A}, t_\psi \rangle}(\pi)$ by $\textsc{Sls}(\pi, \hat{A}, t_\psi)$ (Single-Loop Summary).

Corollary 1. *If $\langle \hat{A}, t \rangle$ is a correct over-approximation of $\langle A, \tau \rangle$, then* $\textsc{Sls}(\pi, \hat{A}, t_\psi)$ *is a correct over-approximation of π.*

Example 4. We continue the running example. Recall the abstract domain in Ex. 2. Let π' denote the loop body of the example program and let q denote the loop guard. By applying the symbolic transformer from Ex. 3, we can check that the following conditions hold:

1. $MOP_{\pi'}(t, t(q)(\phi)) \sqsubseteq \phi$ for any assertion $((p_a \leq l_a) \wedge z_a \wedge \neg s_a)$.
2. $MOP_{\pi'}(t, t(q)(\phi)) \sqsubseteq \phi$ for the assertion z_a.

Thus, we summarize the loop with the following predicate transformer:

$$(z_a \rightarrow z'_a) \wedge (((p_a \leq l_a) \wedge z_a \wedge \neg s_a) \rightarrow ((p'_a = l'_a) \wedge z'_a \wedge \neg s'_a)) \; .$$

3.4 Summarization for Arbitrary Programs

We now describe an algorithm for over-approximating an arbitrary program. Like traditional algorithms (e.g. [19]), the dependency tree of program fragments is traversed bottom-up, starting from the leaves. The code fragments we consider may be function calls or loops. We treat function calls as arbitrary sub-graphs (see Def. 2) of the program graph, and do not allow recursion. We support irreducible graphs using loop simulation [20].

Specifically, we define the *sub-graph dependency tree* of a program $\pi = \langle U, G \rangle$ as the tree $\langle T, > \rangle$, where

- the set of nodes of the tree are program sub-graphs of G;
- for $G_1, G_2 \in T$, $G_1 > G_2$ iff G_2 is a program sub-graph of G_1 with $G_1 \neq G_2$;
- the root of the tree is G;
- every leaf is a loop-free or single-loop sub-graph;
- every loop sub-graph is in T.

```
1  SUMMARIZE(π)
   input  : program π = ⟨U, G⟩
   output : over-approximation π′ of π
2  begin
3      ⟨T, >⟩ :=sub-graph dependency tree of π;
4      π_r := π;
5      for each G′ such that G > G′ do
6          ⟨U, G″⟩:=SUMMARIZE(⟨U, G′⟩);
7          π_r := π_r where G′ is replaced with G″;
8          update ⟨T, >⟩;
9      if π_r is a single loop then
10         ⟨Â, t⟩ := choose abstract interpretation for π_r;
11         ψ := test invariant candidates for t on π_r;
12         π′ := SLS(π_r, Â, t_ψ);
13     else
           /* π_r is loop-free */
14         π′ := Sum_⟨A,τ⟩(π_r);
15     return π′
16 end
```

Algorithm 1: Generic program summarization

Algorithm 1 takes a program as input and computes its summary by following the structure of the sub-graph dependency tree (Line 3). Thus, the algorithm is called recursively on the sub-program until a leaf is found (Line 5). If it is a single loop, an abstract domain is chosen (Line 10) and the loop is summarized as described in Section 3.3 (Line 12). If it is a loop-free program, it is summarized with a symbolic transformer as described in Section 3.2 (Line 14). The old sub-program is then replaced with its summary (Line 7) and the sub-graph dependency tree is updated (Line 8). Eventually, the entire program is summarized.

Theorem 2. SUMMARIZE(π) *is a correct over-approximation of* π.

The precision of the over-approximation is controlled by the precision of the symbolic transformers. However, in general, the computation of the best abstract transformer is an expensive iterative procedure. We use the inexpensive syntactic procedure for loop-free fragments. Loss of precision only happens when summarizing loops, and greatly depends on the abstract interpretation chosen in Line 10.

Note that Algorithm 1 does not limit the selection of abstract domains to any specific type of domains, and that it does not iterate the predicate transformer on the program. Furthermore, this algorithm allows for *localization* of the summarization procedure, as a new domain may be chosen for every loop. Once the domains are chosen, it is also easy to monitor the progress of the summarization, as the number of loops and the cost of computing the symbolic transformers are known – another distinguishing feature of our algorithm.

The summarization can serve as an over-approximation of the program. It can be trivially analyzed to prove unreachability, or equivalently, to prove assertions.

3.5 Leaping Counterexamples

Let π' denote the summary of the program. The program π' is a loop-free sequence of symbolic summaries for loop-free fragments and loop summaries. A *counterexample* for an assertion in π' follows this structure: when traversing symbolic summaries for loop-free fragments, it is identical to a concrete counterexample. Upon entering a loop summary, the effect of the loop body is given as a single transition in the counterexample: we say that the counterexample *leaps* over the loop.

Example 5. Consider the summary from Ex. 4. Suppose that in the initial condition, the buffer a contains a null terminating character in position n and no $'/'$ character. If we check that, after the loop, p_a is greater than the size n, we obtain a counterexample with $p_a^0 = 0, p_a^1 = n$.

The *leaping counterexample* may only exist with respect to the abstract interpretations used to summarize the loops, i.e., they may be spurious in the concrete interpretation. Nevertheless, they provide useful diagnostic feedback to the programmer, as they show a (partial) path to the violated assertion, and contain many of the input values the program needs to read to violate the assertion. Furthermore, spurious counterexamples can be eliminated by combining our technique with counterexample-guided abstraction refinement, as we do have an abstract counterexample.

4 Experimental Evaluation

We implemented our loop summarization technique in a tool called LoopFrog and report our experience using abstract domains tailored to the discovery of buffer overflows on a large set of ANSI-C benchmarks.[1] The loop summarization (as described in Section 3.3) relies on the symbolic execution engine of CBMC. We use bit-blasting to SAT as a decision procedure, but any SMT-BV solver is applicable as well.

We use Goto-CC[2] to extract model files from C source code; full ANSI-C is supported. The model files essentially contain a symbol table and a control flow graph. LoopFrog performs a field-sensitive pointer analysis, which is used to add assertions about pointers. The program is then passed to the loop summarization and, finally, the CBMC assertion checker.

The current implementation of LoopFrog is able to automatically check user-supplied assertions of arbitrary form. In addition to these, array and dynamic object bounds checks, pointer validity, and string termination assertions

[1] Note that our technique is a general-purpose loop and function call summarization method. It is not limited to special classes of faults such as buffer overflows.

[2] http://www.cprover.org/goto-cc/

are added automatically, where required. Also, abstract models for string-related functions from the ANSI-C library are provided and added if necessary.

4.1 An Abstract Domain for Strings

The choice of the abstract domain for the loop summarization has a significant impact on the performance of the algorithm. A carefully selected domain generates fewer invariant candidates and thus speeds up the computation of a loop summary. Besides, the abstract domain has to be sufficiently expressive to retain enough of the semantics of the original loop to show the property.

In order to evaluate the effectiveness of the summarization algorithm, we use programs that manipulate string buffers as benchmarks. We therefore implement the following string-related abstract domain, similar to the instrumentation suggested by Dor et al. [21]: for each string buffer s, a Boolean value z_s and integers l_s and b_s are tracked. The Boolean z_s holds if s contains the zero character within the buffer size b_s. If so, l_s is the index of the first zero character, otherwise, l_s has no meaning.

In our experiments, we use the following assertions for the abstract states, which we call *invariant templates*:

- Arithmetic relations between i and l_s, where i is an integer type expression, and s is a string buffer. Currently, we use $0 \leq i < l_s$.
- String termination, i.e., z_s holds.
- String length, i.e., $l_s < b_s$ holds.
- Pointer validity: p points to a specific object. Currently, we use the weaker $p \neq \mathit{NULL}$.

These templates are instantiated according to variables occurring in the code fragment taken into account. To lower the amount of template instantiations, the following set of simple heuristics is used:

1. Only variables of appropriate type are considered (we concentrate on string types).
2. Indices and string buffers are combined in one invariant only if they are used in the same expression, i.e., we detect instructions which contain $p[i]$ and build invariants that combine i with all string buffers pointed by p.

These templates have proven to be effective in our experiments. Other applications likely require different abstract domains. However, new domain templates may be added quite easily: they usually can be implemented with less than a hundred lines of code.

4.2 Results on Small Benchmarks

We use metrics proposed by Zitser et al. [22] to evaluate and compare the precision of our implementation. We report the *detection rate* $R(d)$ and the *false positive rate* $R(f)$. The *discrimination rate* $R(\neg f|d)$ is defined as the ratio of

test cases on which an error is correctly reported, while it is, also correctly, not reported in the corresponding fixed test case. Using this measure, tools are penalized for not finding a bug, but also for not reporting a fixed program as safe.

The experiments are performed on two recently published benchmark sets. The first one, by Zitser et al. [22], contains 164 instances of buffer overflow problems, extracted from the original source code of `sendmail`, `wu-ftpd`, and `bind`. The test cases do not contain complete programs, but only those parts required to trigger the buffer overflow. According to Zitser et al., this was necessary because the tools in their study were all either unable to parse the test code, or the analysis used disproportionate resources before terminating with an error ([22], pg. 99). In this set, 82 tests contain a buffer overflow, and the rest represent a fix of a buffer overflow.

The results of a comparison with a wide selection of static analysis tools[3] are summarized in Table 1. Almost all of the test cases involve array bounds violations. Even though Uno, Archer and BOON were designed to detect these type of bugs, they hardly report any errors. The source code of the test cases was not annotated, but nevertheless, the annotation-based Splint tool performs reasonably well on these benchmarks. LoopFrog is the only tool that reports all buffer overflows correctly (a detection rate of $R(d) = 1$) and with 62%, LoopFrog also has the highest discrimination rate among all the tools. It is also worth noticing that our summarization technique performs quite well, when only a few relational domains are used (the second line of Table 1). The third line in this table contains the data for a simple interval domain, not implemented in LoopFrog, but as a traditional abstract domain; it reports almost everything as unsafe.

	$R(d)$	$R(f)$	$R(\neg f \mid d)$
LoopFrog	1.00	0.38	0.62
$=, \neq, \leq$	1.00	0.44	0.56
Interval Domain	1.00	0.98	0.02
Polyspace	0.87	0.50	0.37
Splint	0.57	0.43	0.30
Boon	0.05	0.05	0
Archer	0.01	0	0
Uno	0	0	0
LoopFrog [23]	1.00	0.26	0.74
$=, \neq, \leq$[23]	1.00	0.46	0.54

Table 1. Effectiveness: Detection rate $R(d)$, false positive rate $R(f)$, and discrimination rate $R(\neg f \mid d)$ for various static analysis tools.

The second set of benchmarks was proposed by Ku et al. [23]. It contains 568 test cases, of which 261 are fixed versions of buffer overflows. This set partly overlaps with the first one, but contains source code of a greater variety of applications, including the Apache HTTP server, Samba, and the NetBSD C system library. Again, the test programs are stripped down, and are partly simplified to enable current model checkers to parse them. Our results on this set confirm the results obtained using the first set; the corresponding numbers are given in the last two lines of Table 1. On this set the advantage of selecting property-specific

[3] The data for all tools but LoopFrog, "$=, \neq, \leq$" and the Interval Domain is from [22].

domains is clearly visible, as a 20% increase in the discrimination rate over the simple relational domains is witnessed. Also, the performance of LoopFrog is much better if specialized domains are used, simply because there are fewer candidates for the invariants.

The leaping counterexamples computed by our algorithm are a valuable aid in the design of new abstract domains that decrease the number of false positives. Also, we observe that both test sets include instances labelled as unsafe that LoopFrog reports to be safe (1 in [22] and 9 in [23]). However, by manual inspection of the counterexamples for these cases, we find that our tool is correct, i.e., that the test cases are spurious.[4] For most of the test cases in the benchmark suites, the time and memory requirements of LoopFrog are negligible. On average, a test case finishes within a minute.

4.3 Large-Scale Benchmarks

We also evaluated the performance of LoopFrog on a set of large-scale benchmarks, that is, complete un-modified program suites. Table 2 contains a selection of the results.[5] Further experimental data, an in-depth description of LoopFrog, the tool itself, and all our benchmark files are available on-line for experimentation by other researchers.[6] Due to the problems reported by Zitser et al., we were unable to apply other tools to the large-scale benchmarks.

These experiments clearly show that the algorithm scales reasonably well in both memory and time, depending on the program size and the number of loops contained. The time required for summarization naturally depends on the complexity of the program, but also to a large degree on the selection of (potential) invariants. As experience has shown, unwisely chosen invariant templates may generate many useless potential invariants, each requiring to be tested by the SAT-solver. This is a problem that we seek to remedy in the future, by leveraging incremental SAT-solver technology.

In general, the results regarding the program assertions shown to hold are not surprising; for many programs (e.g., texindex, ftpshut, ginstall), our selection of string-specific domains proved to be quite useful. It is also interesting to note that the results on the ftpshut program are very different on program versions 2.5.0 and 2.6.2: This program contains a number of known buffer-overflow problems in version 2.5.0, and considerable effort was spent on fixing it for the 2.6.2 release; an effort clearly reflected in our statistics. Just like in this benchmark, many of the failures reported by LoopFrog correspond to known bugs and the leaping counterexamples we obtain allow us to analyze those faults. Merely for reference we list CVE-2001-1413 (a buffer overflow in ncompress) and CVE-2006-1168 (a buffer underflow in the same program), for which we are easily able to produce counterexamples.[7] On the other hand, some other programs (such as the ones

[4] We exclude those instances from our benchmarks.

[5] All data was obtained on an 8-core Intel Xeon 3.0 GHZ. We limited the runtime to 4 hours and the memory per process to 4 GB.

[6] http://www.cprover.org/loopfrog/

[7] The corresponding bug reports may be obtained from http://cve.mitre.org/.

Suite	Program	Instructions	# Loops	Time			Peak Memory	Assertions		
				Summarization	Checking Assertions	Total		Total	Passed	Violated
freecell-solver	aisleriot-board-2.8.12	347	26	10s	295s	305s	111MB	358	165	193
freecell-solver	gnome-board-2.8.12	208	8	0s	3s	4s	13MB	49	16	33
freecell-solver	microsoft-board-2.8.12	168	4	2s	9s	11s	32MB	45	19	26
freecell-solver	pi-ms-board-2.8.12	185	4	2s	10s	13s	33MB	53	27	26
gnupg	make-dns-cert-1.4.4	232	5	0s	0s	1s	9MB	12	5	7
gnupg	mk-tdata-1.4.4	117	1	0s	0s	0s	3MB	8	7	1
inn	encode-2.4.3	155	3	0s	2s	2s	6MB	88	66	22
inn	ninpaths-2.4.3	476	25	5s	40s	45s	49MB	96	47	49
ncompress	compress-4.2.4	806	12	45s	4060s	4106s	345MB	306	212	94
texinfo	ginstall-info-4.7	1265	46	21s	326s	347s	127MB	304	226	78
texinfo	makedoc-4.7	701	18	9s	6s	16s	28MB	55	33	22
texinfo	texindex-4.7	1341	44	415s	9336s	9757s	1021MB	604	496	108
wu-ftpd	ckconfig-2.5.0	135	0	0s	0s	0s	3MB	3	3	0
wu-ftpd	ckconfig-2.6.2	247	10	13s	43s	57s	27MB	53	10	43
wu-ftpd	ftpcount-2.5.0	379	13	10s	32s	42s	37MB	115	41	74
wu-ftpd	ftpcount-2.6.2	392	14	8s	24s	32s	39MB	118	42	76
wu-ftpd	ftprestart-2.6.2	372	23	48s	232s	280s	55MB	142	31	111
wu-ftpd	ftpshut-2.5.0	261	5	1s	9s	10s	13MB	83	29	54
wu-ftpd	ftpshut-2.6.2	503	26	27s	79s	106s	503MB	232	210	22
wu-ftpd	ftpwho-2.5.0	379	13	7s	23s	30s	37MB	115	41	74
wu-ftpd	ftpwho-2.6.2	392	14	8s	27s	35s	39MB	118	42	76
wu-ftpd	privatepw-2.6.2	353	9	4s	17s	22s	32MB	80	51	29

Table 2. Large-Scale Evaluation.

from the freecell-solver suite) clearly require different abstract domains, suitable for other heap structures than strings. The development of suitable domains and subsequent experiments, however, are left for future research.

5 Conclusion and Future Work

We presented a novel algorithm for program verification using symbolic abstract transformers. The algorithm computes an abstraction of a program with respect to a given abstract interpretation by replacing loops and function calls in the control flow graph by their symbolic transformers. The runtime of our algorithm is linear in the number of looping constructs. It addresses the perennial problem of the high complexity of computing abstract fixpoints. The procedure over-approximates the original program, which implies soundness of our analysis. An additional benefit of the technique is its ability to generate *leaping counterexamples*, which are helpful for diagnosis of the error or for filtering spurious warnings. Experimental results show the best error-detection and error-discrimination rates comparing to a broad selection of static analysis tools. As future work, we plan to analyze the leaping counterexamples automatically in order to rule out spurious traces and to refine the abstract domain.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
2. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints Among Variables of a Program. In: POPL. (1978) 84–96
3. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: PLILP. LNCS, Springer (1992) 269–295
4. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: VMCAI. LNCS, Springer (2004) 252–266
5. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. FMSD **25** (2004) 105–127
6. Gopan, D., Reps, T.W.: Low-level library analysis and summarization. In: CAV. LNCS, Springer (2007) 68–81
7. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. Program Flow Analysis: theory and applications. Prentice-Hall (1981)
8. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the Saturn project. In: PASTE, ACM (2007) 43–48
9. Babic, D., Hu, A.J.: Calysto: scalable and precise extended static checking. In: ICSE, ACM (2008) 211–220
10. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: ISSTA. (2000) 14–25
11. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, Springer (2004) 168–176
12. Clarke, E., Grumberg, O., Peled, D.A.: Model checking. MIT Press (1999)
13. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. (1979) 269–282
14. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. LNCS, Springer (1997) 72–83
15. Lahiri, S.K., Ball, T., Cook, B.: Predicate abstraction via symbolic decision procedures. In: CAV. LNCS, Springer (2005) 24–38
16. Kroening, D., Sharygina, N.: Approximating predicate images for bit-vector logic. In: TACAS. LNCS, Springer (2006) 242–256
17. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: CAV. LNCS, Springer (2006) 424–437
18. Cavada, R., Cimatti, A., Franzén, A., Kalyanasundaram, K., Roveri, M., Shyamasundar, R.K.: Computing predicate abstractions by integrating BDDs and SMT solvers. In: FMCAD, IEEE (2007) 69–76
19. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28** (1981) 594–614
20. Ashcroft, E., Manna, Z.: The translation of 'go to' programs to 'while' programs. (1979) 49–61
21. Dor, N., Rodeh, M., Sagiv, S.: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In: PLDI. (2003) 155–167
22. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. In: SIGSOFT FSE. (2004) 97–106
23. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE '07, ACM Press (2007) 389–392