SMT-Based False Positive Elimination in Static Program Analysis

Maximilian Junker[1], Ralf Huuck[2], Ansgar Fehnker[2], and Alexander Knapp[3]

[1] Technische Universität München, Munich, Germany
junkerm@in.tum.de
[2] NICTA, University of New South Wales, Sydney, Australia
{ansgar.fehnker,ralf.huuck}@nicta.com
[3] Universität Augsburg, Augsburg, Germany
knapp@informatik.uni-augsburg.de

Abstract. Static program analysis for bug detection in large C/C++ projects typically uses a high-level abstraction of the original program under investigation. As a result, so-called false positives are often inevitable, i.e., warnings that are not true bugs. In this work we present a novel abstraction refinement approach to automatically investigate and eliminate such false positives. Central to our approach is to view static analysis as a model checking problem, to iteratively compute infeasible sub-paths of infeasible paths using SMT solvers, and to refine our models by adding observer automata to exclude such paths. Based on this new framework we present an implementation of the approach into the static analyzer Goanna and discuss a number of real-life experiments on larger C code projects, demonstrating that we were able to remove most false positives automatically.

1 Introduction

Static program analysis of industrial size C/C++ programs for the detection of quality as well as security bugs has had some considerable success in the recent years. A number of software tools and companies [23,12] resulted from theoretical advances and increased computing power, leading to the detection of complex source code defects with minimal effort from the side of the developers.

However, static analysis techniques are based on approximations of the original source code semantics. As such, the results of static analyzers might contain spurious warnings, i.e., false positives. The task of assessing the validity of tool warnings falls back to the developer. But with the increasing complexity of the bugs that those techniques can uncover, this assessment is getting more and more difficult. In large software projects developers may be forced to spend a lot of time reconstructing a warning of a static analysis tool just to discover that the claimed bug is not real. Therefore, it is vital for static analysis tools not only to find many complex bugs, but also to assure that the majority of those are not false positives.

Unlike static program analysis, traditional software model checking has established methods in dealing with abstractions and false positives, which are referred to as spurious counter-examples. One particular prominent method is counter-example guided abstraction refinement (CEGAR) [6]. In the world of static program analysis, however, there is no good notion of automatic iterative refinement. Moreover, CEGAR approaches typically refine in each iteration the whole program/function under consideration and re-run the analysis on the new model, which can often be costly.

In this work we adopt some ideas from such established techniques, but take a significantly different approach. The individual key insights and contributions are:

1. We define static program analysis problems in terms of syntactic model checking problems. In this context a bug is a violation of a syntactic model checking formula resulting in a counter-example.
2. We symbolically evaluate the feasibility of such a counter-example on a low-level program semantics using an SMT solver. If the counter-example path is infeasible we compute slices of this path that are the cause for its infeasibility and we construct an observer automaton that excludes all paths with the same cause.
3. Unlike in CEGAR we do *not* refine the whole model, but only add the observer automaton to the original model. We repeat the procedure until either all counter-examples are eliminated or a bug is found that could not be eliminated.

We evaluate our approach by applying it to a number of case studies from the NIST SAMATE program [23] and show that most of the relevant false positives can be efficiently removed using the proposed method.

Outline. In Sect. 2 we give a high-level introduction and overview of our model checking approach to static analysis as well as the ideas of the refinement loop using observers to exclude infeasible paths. We provide more details on computing infeasible sub-paths in Sect. 3 and on the construction of the observers for language refinement in Sect. 4. This is followed by large scale experiments in Sect. 5. Related work is discussed in Sect. 6. Finally, we conclude with an outlook to future work in Sect. 7.

2 Syntactic Model Checking and Language Refinement

In this section we describe our model checking approach to static program analysis and explain the key concepts of our false-positive elimination procedure. The idea of using model checking for static program analysis has first been introduced by Steffen and Schmidt [24], discussing how data flow analysis problems can be expressed in modal μ-calculus. This has later been expanded and further developed in [18,8,19].

The main idea is to abstractly represent a program (or a single function) by its control flow graph (CFG) annotated with labels representing propositions of interest. Example propositions are whether memory is allocated or freed in a particular location, whether a pointer variable is assigned $null$ or whether it is dereferenced. In this way the possibly infinite state space of a program is reduced to the finite set of locations and their propositions.

The annotated CFG consisting of the transition system and the (atomic) propositions can then be transformed into the input language of a model checker. Static analysis bug patterns can be formulated in a temporal logic and evaluated automatically by the model checker. As the annotated CFG discards most of the program semantics apart

```
void foo() {
l0 : int x, *a;
l1 : int* p = malloc(sizeof(int));
     for(l2 : x = 10; l3 : x > 0; l7 : x--) {
     l4 : a = p;
     l5 : if(x == 1)
          l6 : free(p);
     }
}
```

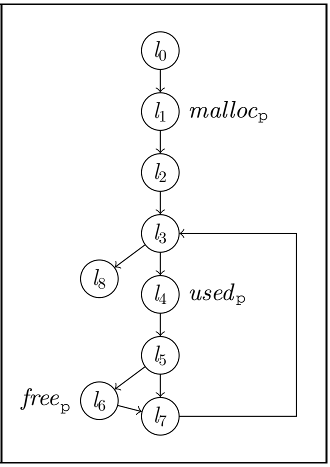

Fig. 1. Example of an annotated CFG for a function `foo`. The locations are also annotated in the listing.

from the annotations and reduces a program to its syntactical structure the approach is called *syntactic model checking* [13].

To illustrate the approach, we use a contrived function `foo` shown in Fig. 1. It works as follows: First a pointer variable `p` is initialized and memory is allocated accordingly. Then, in a loop, a second pointer variable `a` is assigned the address saved in `p`. After the tenth assignment `p` is freed and the loop is left.

To automatically check whether the memory allocated for `p` is still accessed after it is freed (a *use-after-free* in static analysis terms) we define atomic propositions for allocating memory $malloc_{\mathrm{p}}$, freeing memory $free_{\mathrm{p}}$ and accessing memory $used_{\mathrm{p}}$, and we label the CFG accordingly. The above check can now be expressed in CTL as:

$$\mathbf{AG}(malloc_{\mathrm{p}} \Rightarrow \mathbf{AG}(free_{\mathrm{p}} \Rightarrow \mathbf{AG}\neg used_{\mathrm{p}}))$$

This means, whenever memory is allocated, after a $free_{\mathrm{p}}$ there is no occurrence of a $used_{\mathrm{p}}$. Note that once a check has been expressed in CTL, the proposition can be generically pre-defined as a template of syntactic tree patterns on the abstract syntax tree of the code and determined automatically. Hence, it is possible to automatically check a wide range of programs for the same requirement. However, since our approach for false-positive elimination is based on checking path-infeasibility, we are restricted to formulas that allow linear counter-examples.

2.1 False-Positive Detection

Model checking the above property for the model depicted in Fig. 1 will find a violation and return a counter-example. The following path denoted by the sequence of locations is such a counter-example: $l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_3, l_4, l_5$.

However, if we match up the counter-example in the abstraction with the concrete program, we see that this path cannot possibly be executed, as the condition `x == 1`

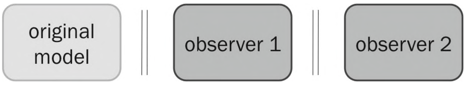

Fig. 2. Parallel composition of observers with original model

cannot be true in the first loop iteration and, therefore, l_5 to l_6 cannot be taken. This means, the counter-example is spurious and should be discarded. We might get a different counter-example in the last loop iteration $\ldots, l_5, l_6, l_7, l_3, l_4, l_5$. But again, such a counter-example would be spurious, because once the condition `x == 1` holds, the loop condition prevents any further iteration.

To detect the validity of a counter-example we subject the path to a fine-grained simulation using an SMT solver. In essence, we perform a backward simulation of the path computing the *weakest precondition*. If the precondition for the initial state of the path is unsatisfiable, the path is infeasible and the counter-example spurious.

2.2 Observer Computation

Once we identified a counter-example as being spurious we know that this particular path is infeasible, but that does not mean that there are no other counter-examples for the same property. Therefore, we need to rerun the check on a refined model to see if there are other counter-examples. To get a refined model we construct a set of observer automata that have the following properties:

1. The observers can be run with the original abstract model, but they restrict the abstract model by excluding the previously computed infeasible paths.
2. The observers are based on the minimal infeasible sub-paths of a counter-example. This means, we do not need to encode each infeasible path individually, but only the set of statements that are unsatisfiable. As an example consider the assignment `x = 10` and the condition `x == 1`. Any path through these two statements, and not modifying `x` in between, will be infeasible. Hence, an observer monitoring the sub-path can be sufficient for ruling out many paths simultaneously.

Figure 2 schematically illustrates the idea of running the original model with a set of observers each representing a minimal reason for paths being infeasible. We require that in the newly composed model no observer can reach its final state, i.e., all infeasible sub-paths are excluded.

2.3 Refinement Loop

After constructing the observers based on the infeasible sub-paths, the original abstract model can be rerun to see if there are other possible counter-examples. The full path refinement loop is presented in Fig. 3. The refinement loop successively constructs new observers for new infeasible paths and extends the original model accordingly. There are two termination conditions: Firstly, we terminate whenever no bug in a program is found, i.e., there is no counter-example in the (extended) model. Secondly, we terminate

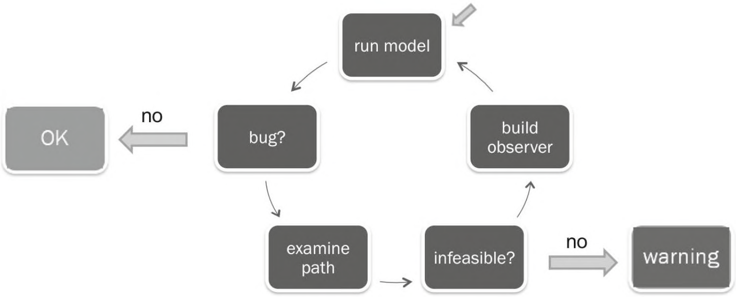

Fig. 3. Counter-example guided path refinement loop

when a path cannot be discharged as infeasible. There are two reasons for the latter: Either we found a genuine bug or our program semantics encoded in the SMT solver does not model certain aspects that are necessary to dismiss a path.

There are a few things worth noting: As we will see in the subsequent section we cover a wide variety of aspects in the C semantics including pointer aliasing. However, some constructs such as function pointers are not taken into account. In our experience, these program constructs are rarely the cause of false positives. Moreover, in the worst case we have to construct one observer for each infeasible path and there might be an exponential number of infeasible paths w.r.t. the number of conditional statements in a program. In practice, we found the number of required observers to be quite small. For most real-life cases the abstraction refinement loop terminates after two or three iterations.

2.4 A Word on SMT Solvers

In general, SMT solving tackles the satisfiability of first-order formulae modulo background theories. The approach presented in this paper is largely independent of the particular SMT solver used. However, in order to represent our semantic model of C, we require a minimum set of theories including uninterpreted functions, linear integer arithmetic and the theory of arrays, and we consider the SMT solver to support infeasible core computation.

Using additional theories (such as bit-vectors) can improve the overall precision of the presented approach for a potential penalty in runtime. We discuss the results with the given sets of theories in Sect. 5.

3 Computing Reasons for Infeasible Paths

For the path reduction refinement loop we have to identify infeasible paths. Moreover, we are interested in a small sequence of statements that explains why a path is infeasible. Such an explanation will allow us to exclude all paths with that infeasible sequence of statements. For instance, in the path through $l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_3, l_4$ of `foo` in Fig. 1 not all statements are contributing to it being infeasible, but only l_2 : `x = 10` and (l_5, l_6) : `x == 1`. We call the sequence of edges $(l_2, l_3), (l_3, l_4), (l_4, l_5), (l_5, l_6)$ an *infeasible sub-path.*

Next, we explain how to detect infeasible paths in a C program by means of satisfiability checking of weakest preconditions using an SMT solver. Moreover, we provide a strategy to efficiently compute an infeasible sub-path from an infeasible path, which enables the construction of an efficient observer.

3.1 Detecting Infeasible Paths

For checking the feasibility of a path we first collect the sequence of statements in that path along the CFG. Moreover, we encode branching decisions along that path as assertions in the sequence of statements, resulting in a straight-line program. Next, we compute the weakest precondition for this straight-line program. The SMT solver might return that the weakest precondition is unsatisfiable, i.e., that the path cannot be executed and, therefore, is infeasible. In the following we provide the basic ideas for modeling program statements, their semantics and the representation in an SMT solver. The details of the underlying semantics and the memory model can be found in [21].

Path programs. Computing the straight-line program corresponding to a path through the CFG amounts to collecting the sequence of assignments and function calls taken on the path. Additionally, assert statements record what must be true at a point of execution to follow the path through control-flow statements, like taking the then-branch of an `if`. For example, the path $(l_2, l_3), (l_3, l_4), (l_4, l_5), (l_5, l_6), (l_6, l_7)$ of the CFG of `foo` in Fig. 1 is represented by the path program

```
x = 10; assert(x > 0); a = p; assert(x == 1); free(p);
```

More formally, a *path program* is a sequence of statements s containing expressions e. A statement can be e_1 `=` e_2`;` for assignments, assert(e)`;` for checking a (boolean) condition and f`(`e_1`, ... ,` e_n`);` respectively e_0 `=` f`(`e_1`, ... ,` e_n`);` for function calls (optionally assigning the return value). In our approach, an expression may be almost any valid C-expression [20] including pointers and using `structs`. Currently, however, we do not support function pointers, and string literals are treated as fresh pointer variables. We make the simplifying assumption that identifiers such as program variables and field names of `structs` are globally unique.

Weakest precondition. The set of states from which a path program can be executed without violating any assertion is given by the weakest precondition of the path program w.r.t. the trivial postcondition $true$. Generally, the *weakest precondition* $wp(p, \psi)$ of a path program p w.r.t. a condition ψ on states is given by a condition φ which is satisfied by exactly those states from which an execution of p terminates in a state satisfying ψ. In particular, $wp($assert(e)`;`$, \psi)$ is equivalent to $e \wedge \psi$, indeed asserting that e must already hold.

The computed formula $wp(p, true)$ characterizing successful executability of p will be handed to an SMT solver for checking unsatisfiability. However, we will not always be able to represent executability faithfully in terms of a full C-semantics, but may have to use safe approximations. These approximations have to ensure (under certain assumptions) that the unsatisfiability of $wp(p, \mathit{true})$ implies that p is not executable.

In particular, our definition of wp is based on a simple memory model that allows a good precision even in the presence of variable aliasing and basic pointer arithmetic. The memory model is similar to the one described by Burstall [5]. The main idea is to have a separate store, represented by a separate variable, for each primitive data type. We use a simple type system consisting of primitive types like integers and pointer types as well as `struct`-like composite types. Each store is again segmented into partitions, one for each field of a `struct` and a distinguished partition for data that is not part of a `struct`. The rationale behind this kind of model is that by introducing logical structure such as distinct memories we achieve the property that certain aliasing is not possible, such as aliasing between variables of different types or between different fields of a `struct`. Properties that we do not get by construction are enforced by axioms. An example for such an axiom is that local variables do not alias.

The memory model can be easily encoded in a language for SMT solvers. In our experiments we used the theory of arrays [25]. The theory provides two operations: $\mathit{access}(m, a)$ (sometimes called *read* or *select*) to access the value of an array m at location a and $\mathit{update}(m, a, v)$ (sometimes called *write* or *store*) to get a version of m that is updated at location a with value v. Using the theory of arrays, the memory model can be represented as an array with tuples (containing a partition name and a position) as indices.

To illustrate the use of the memory model in the wp-semantics we consider a simple assignment `x = v`, where the value of local variable `x` of type τ is assigned the value of a local variable `v`, also of type τ. The wp-semantics in this case is

$$wp(\mathtt{x = v}, \varphi) = \varphi\{M_\tau \mapsto \mathit{update}(M_\tau, \mathit{loc}(\mathtt{x}), v)\} \wedge v = \mathit{access}(M_\tau, \mathit{loc}(\mathtt{v}))$$

where M_τ is the memory variable for τ and loc is a function mapping a variable name to a location (i.e. partition and position).

A limitation with regard to the definition of wp are function calls. We currently do not consider the true effect of called functions, but approximate the effect by assuming that only those locations in the memory are touched that are explicitly passed as a pointer. An exception in this regard is the `malloc` function. As it is central to handle pointers sufficiently precise we use axioms to specify its semantics. An example for such an axiom is that `malloc` always returns fresh memory locations, which are not aliased with any other. On the other hand, no special axioms are needed for `free`, ignoring whether its argument points to allocated memory. More details on the wp-semantics can be found in [21].

Infeasible Paths. Based on the weakest precondition semantics, infeasibility follows naturally as: A path through the CFG is called *infeasible* if $wp(p, \mathit{true})$ for its corresponding path program p is unsatisfiable.

For example, the path $(l_2, l_3), \ldots, (l_6, l_7)$ from above is infeasible since

$$wp(\mathtt{x = 10;\ assert(x > 0);\ a = p;\ assert(x == 1);\ free(p);}, \mathit{true})$$

is unsatisfiable due to incompatibility of `x = 10` and `x == 1`. Next, we explain how to identify shorter sub-paths capturing the relevant causes for infeasible paths.

3.2 Computing Infeasible Sub-paths

The general idea of this work is to create observers to exclude infeasible paths in static program analysis. We would like, however, to avoid to generate one observer for each path, but instead to identify smaller sub-paths that capture unsatisfiable inconsistencies. Excluding these sub-paths might exclude a wider set of paths passing through those fragments and, as a result, one observer will be able to exclude many infeasible paths at once. For instance, the path $(l_2, l_3), \ldots, (l_6, l_7)$ above shows that with respect to infeasibility it is irrelevant how an execution reaches l_2 : `x = 10` and how it continues after (l_5, l_6) : `assert(x == 1);`: Due to the incompatibility of `x = 10` and `x == 1` on that path, any path containing $(l_2, l_3), (l_3, l_4), (l_4, l_5), (l_5, l_6)$ as a sub-path is infeasible.

Generally, let us say that path π' is a *sub-path* of path π and π *includes* π' if π is of the form $\pi_0\ \pi'\ \pi_1$ for some (possibly empty) paths π_0 and π_1. This leads to:

Proposition 1. *Every path including an infeasible sub-path is infeasible.*

Thus, if we find an infeasible sub-path in a path from a counter-example, we can exclude all paths that include this sub-path. We can compute the infeasible sub-paths by computing the unsatisfiable sub-formulae of the weakest precondition. In order to match sub-formulae and sub-paths we split the weakest precondition into named conjuncts each of which corresponds to a statement in the considered path (see [21] for details).

SMT solvers usually can be instructed to deliver an unsatisfiable sub-formula. It is advantageous to identify small unsatisfiable sub-formulae leading to short infeasible sub-paths, thus allowing to exclude potentially more paths. However, finding all minimal unsatisfiable sub-formulae requires exponentially many calls to the SMT solver in the worst case (for algorithms see, e.g., [9] and [22]). We therefore heuristically enumerate unsatisfiable sub-formulae using the solver and employ an exponential algorithm only to minimize these [21].

4 Observer Construction and Refinement

In this section we formally define how to construct observers based on sub-paths. Moreover, we show how to compose the observers with the original model in a refinement loop for eliminating false positives.

In short, for the observer construction we view a CFG as a finite automaton that accepts paths as sequences of edges through the CFG as words. From an infeasible sub-path we construct an “observing” finite non-deterministic automaton. The language of this observing automaton is the set of paths which include the infeasible sub-path. We consider the synchronous product of the CFG automaton and the complemented observing automaton, where synchronization is on the shared alphabet, i.e., the edges. This product automaton accepts exactly those paths as sequences of edges that do not show an infeasible sub-path.

4.1 Representing Programs as Automata

We rely on the conventional notion of a finite (non-deterministic) automaton $M = (A, S, T, I, F)$ consisting of an alphabet A, a finite set of states S, a transition relation

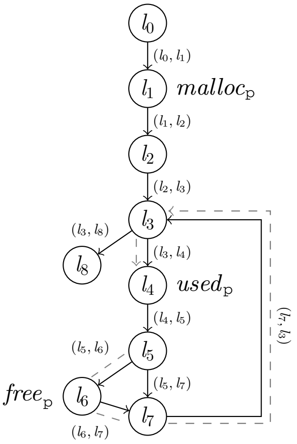

Fig. 4. CFG automaton for the function `foo`. The dashed edges represent an infeasible sub-path.

$T \subseteq S \times A \times S$, a set of initial states $I \subseteq S$, and a set of final states $F \subseteq S$. The words accepted by M are denoted by $\mathcal{L}(M)$. We write $M \times N$ for the synchronous product of the finite automata M and N over the same alphabet; then $\mathcal{L}(M \times N) = \mathcal{L}(M) \cap \mathcal{L}(N)$ holds. The finite automaton yielding language complement is denoted by M^{c}, i.e., $\mathcal{L}(M^{\mathrm{c}}) = A^* \setminus \mathcal{L}(M)$ where A is the alphabet of M.

A CFG can naturally be regarded as such a finite automaton with the states being the locations. For the alphabet we choose pairs of locations(l, l'), i.e., making the edges of the CFG "observable". The transition relation of the automaton just follows from the CFG. All the states are both initial and final to capture arbitrary sub-paths in the CFG.

Definition 1 (CFG Automaton). *For a CFG with locations L and edges $E \subseteq L \times L$, its corresponding* CFG automaton *is the finite automaton given by (E, L, T, L, L), where the alphabet is the set of edges E, the states are the locations L, the transition relation is $T = \{(l, (l, l'), l') \mid (l, l') \in E\}$, and all states are both initial and final.*

The words accepted by a CFG automaton correspond exactly to the paths as sequences of control-flow edges through the CFG. Therefore, we will also call these accepted words "paths". The CFG automaton for the function `foo` is shown in Fig. 4.

A CFG automaton can also be directly used for model-checking, as the annotations of the CFG such as $malloc_{\mathrm{p}}$ can be interpreted as predicates over its states. For `foo` we would define $malloc_{\mathrm{p}} \equiv l_1$ or $used_{\mathrm{x}} \equiv l_3 \vee l_5$.

4.2 Computing Observers from Counter-Examples

If the model checking procedure yields a counter-example as a path through a CFG automaton, which is infeasible, we want to exclude this path in further model checking runs. In fact, the notion of infeasible sub-paths allows us to exclude all paths that include some infeasible sub-path due to Prop. 1. Consider, for example, the CFG automaton in Fig. 4. The dashed edges represent an infeasible sub-path $\pi = (l_5, l_6), (l_6, l_7), (l_7, l_3)$,

(l_3, l_4) of an infeasible counter-example reported by the model checker. We can not only exclude π but also a path that represents a two-fold loop iteration and then continues as before. On the other hand, we cannot exclude a path that has (l_5, l_7) instead of $(l_5, l_6), (l_6, l_7)$.

For a sub-path π accepted by the CFG automaton, we construct an automaton that accepts exactly those paths π' for which π is a sub-path. We define:

Definition 2 (Observer). *Let P be a CFG automaton with alphabet E and let $\pi = e_1 \dots e_k$ be a path accepted by P. The* CFG observer automaton *$Obs(E, \pi)$ is the automaton (E, S_{Obs}, T, S_0, F), where*

- *S_{Obs} is the set of states $\{\mathsf{s}_1, \dots, \mathsf{s}_{k-1}\} \cup \{\mathsf{Init}, \mathsf{Infeasible}\}$.*
- *$T \subseteq S_{Obs} \times E \times S_{Obs}$ is the transition relation. A triple (s, e, s') is in the relation if and only if one of the following holds:*
  1. *$s = \mathsf{Init}$ and $s' = \mathsf{Init}$ and $e \neq e_1$*
  2. *$s = \mathsf{s}_i$ and $s' = \mathsf{s}_{i+1}$ and $e = e_{i+1}$ and $1 \leq i \leq k-2$*
  3. *$s = \mathsf{s}_{k-1}$ and $s' = \mathsf{Infeasible}$ and $e = e_k$*
  4. *$s \neq \mathsf{Infeasible}$ and $s' = \mathsf{s}_1$ and $e = e_1$*
  5. *$s = \mathsf{s}_i$ and $s' = \mathsf{Init}$ and $e \in E \setminus \{e_1, e_{i+1}\}$ and $1 \leq i \leq k-1$*
  6. *$s = \mathsf{Infeasible}$ and $s' = \mathsf{Infeasible}$*
- *$S_0 = \{\mathsf{Init}\}$ is the set of initial states.*
- *$F = \{\mathsf{Infeasible}\}$ is the set of final states.*

The rationale for the particular choice of the observer's components is as follows: The states mirror how much of π has already been observed on a run without interruption. When the observer is in state Init, nothing has been observed at all or a part of π has been observed, but then the sequence was interrupted. If the observer is in state Infeasible the whole path π has already been observed, which means no matter how the program model continues, the current run already represents an infeasible path. If the automaton is in state s_i, we know π has been observed until and including e_i. The transition relation reacts to an edge on the run:

1. As long as the initial edge e_1 of π has not been observed, the observer needs to stay in Init.
2. If the observer has already observed the first i edges of π and now observes the next edge e_{i+1} it proceeds one step further, as long as e_{i+1} is not the last edge of π.
3. If the situation is as in (2) but e_{i+1} is the last edge of π, the observer transitions to Infeasible.
4. It may happen that the observer already is in state s_j when another sequence of π starts. Intuitively, π is interrupted by itself. Therefore the observer may transition to s_1 as soon as it observes e_1, even if it is currently in some s_j.
5. If the sequence is interrupted in a different way, the observer returns to Init.
6. As soon as the observer is in state Infeasible, it remains there forever.

Example 1. We illustrate the observer construction with our running example. Regarding the CFG automaton of the function `foo`, a path containing the sub-path $\pi = (l_5, l_6), (l_6, l_7), (l_7, l_3), (l_3, l_4)$ is infeasible. The constructed observer automaton is depicted in

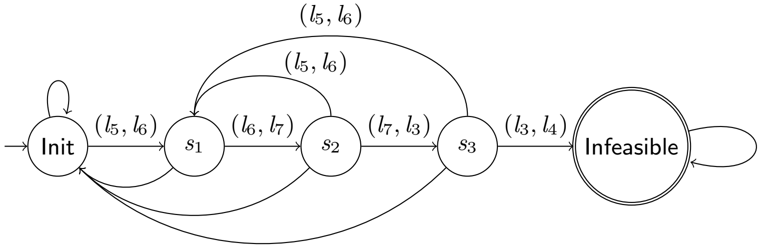

Fig. 5. Observer automaton for infeasible path of Example 1. Unlabeled edges mean "any other".

Fig. 5. As soon as it observes the sequence π it enters the state Infeasible and remains there forever. If the sequence is interrupted, it either returns to Init or, if the interruption equals (l_5, l_6) as the first edge of π, it returns to s_1. Hence, as long as the observer is not in state Infeasible the sequence π has not been observed completely. As Infeasible is the only accepting state, the observer only accepts paths that contain π, i.e., infeasible paths.

Let π denote a path accepted by a CFG automaton P with the alphabet E of control-flow edges and let $\mathcal{P}(\pi)$ be the set of paths in E^* including π. By construction, we have $\mathcal{P}(\pi) = \mathcal{L}(\mathit{Obs}(E, \pi))$, that is, the words accepted by $\mathit{Obs}(E, \pi)$ are exactly the paths including π. Furthermore,

$$\begin{aligned}&\mathcal{L}(P \times \mathit{Obs}(E, \pi)^{\mathrm{c}}) = \mathcal{L}(P) \cap \mathcal{L}(\mathit{Obs}(E, \pi)^{\mathrm{c}}) = \\ &\qquad \mathcal{L}(P) \cap (E^* \setminus \mathcal{L}(\mathit{Obs}(E, \pi))) = \mathcal{L}(P) \cap (E^* \setminus \mathcal{P}(\pi)) \;.\end{aligned}$$

Thus by applying Prop. 1, that all paths including an infeasible sub-path are infeasible, we get

Proposition 2. *Let P be a CFG automaton P with alphabet E and let π be an infeasible path of P. Then the CFG automaton resulting from the synchronous product of P and $\mathit{Obs}(E, w)^{\mathrm{c}}$ excludes the infeasible paths that include π.*

4.3 Implementing Observers

The observer is in general non-deterministic. Computing the complement of a non-deterministic automaton would involve first creating its deterministic equivalent, which can have exponential size compared with the non-deterministic automaton. We avoid directly constructing the complement of the observer and instead implement the complementation by adding a fairness constraint in the model checker [14]. The fairness constraint in our case forbids that the observer enters state Infeasible. Although fair CTL model checking is more complex than regular CTL model checking, it works well in our experiments, as the next section shows.

5 Experiments

In this section we report on the implementation of the aforementioned false-positive elimination techniques as well as on analysis results from representative, large code bases. All the experimental data has been obtained from projects and benchmarks provided by NIST and the Department of Homeland Security for the 2010 and 2011 Static Analysis Tool Exposition (SATE) [23]. The experiments show that the proposed solution provides a significant decrease in false positives while only moderately increasing the overall runtime.

5.1 Implementation

We implemented a prototype of the SMT-based path reduction approach in our static analysis tool Goanna[1]. Goanna is a state-of-the-art static analysis tool for bug detection and security vulnerability analysis of industrial C/C++ programs. Currently, Goanna supports around 150 classes of different checks ranging from memory leak detection and $null$-pointer dereferences to the correct usage of copy control in C++ as well as buffer overruns.

The Goanna tool itself as well as the new false -positive elimination procedure is implemented in the functional programming language OCaml. For the infeasible path detection in our experiments we used the Z3 SMT solver [10], as it provides an OCaml interface which allowed quick prototyping using Goanna.

5.2 Experimental Evaluation

As representative test beds for our experiments we chose the two main open source projects from the NIST SATE 2010 and 2011 exposition: Wireshark 1.2.9 and Dovecot 2.0 beta6. Wireshark is a network protocol analyzer consisting of around $1.4\mathrm{MLoc}$ of pure C/C++ code that expand to roughly $16\mathrm{MLoc}$ after pre-processing (macro expansions, header file inclusion etc.). Dovecot is a secure IMAP and POP3 server that consists of around $170\mathrm{KLoc}$ of pure C/C++ code expanding to $1.25\mathrm{MLoc}$ after preprocessing. We experimented with other in-house industrial code of different sizes as well and obtained very similar results as for the two mentioned projects.

The evaluation was performed on a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at $3.4\mathrm{GHz}$, $2\mathrm{MB}$ L2 cache and $1.5\mathrm{GB}$ DDR-2 $400\mathrm{MHz}$ ECC memory.

False-Positive Removal Rates. As mentioned earlier, Goanna performs a source code analysis for around 150 classes of checks. However, not all checks are path-sensitive, i.e., some checks only require tree-pattern matching, and of those checks that are path-sensitive not all are amenable to false path elimination. The reasons are as follows: Certain path-sensitive checks such as detecting unreachable code already state that there is no path satisfying a certain requirement. Hence, removing infeasible paths will not change the results. A similar example is having no path where a `free()` occurs after a `malloc()` and alike. The results below only include checks where false path elimination can alter the analysis results.

The false-positive elimination results for Wireshark and Dovecot are summarized in Table 1. For Wireshark, our original Goanna implementation detected 98 relevant

[1] `http://www.nicta.com.au/goanna`

Table 1. False Positive Detection Rate for Wireshark and Dovecot

	Wireshark 1.2.9	Dovecot 2.0 beta6
lines of code	1, 368, 222	167, 943
after pre-processing	16, 497, 375	1, 251, 327
number of functions	52, 632	5, 256
issued warnings	98	75
false positives removed	48	38
% removed warnings	49.0%	50.6%
correctly identified false positives	48 (100%)	38 (100%)

Table 2. Runtime Performance for False-Positive Elimination

	Wireshark 1.2.9	Dovecot 2.0 beta6
total running time (no timeout)	8815s	1025s
time spent in refinement loop	1332s (15%)	302s (29.5%)
% of time in SMT	10.5%	12.2%
% of time in model checking	87.5%	86.3%
number of Goanna timeouts	12	1
number of SMT loops exceeding (20)	11	3
number of SMT solver timeouts	0	5

path-sensitive issues. Running Goanna with the new SMT-based false path elimination approach removed 48 issues. This means, around 49% of the produced warnings were eliminated fully automatically. We manually investigated all of the removed warnings and were able to confirm that these were indeed all false positives.

The results for Dovecot are very similar to the Wireshark results. The original implementation raised 75 warnings and we were able to automatically identify 38 of those warnings as false positives. This means, the number of warnings was reduced by 50.6%. Again, all the automatically removed warnings were confirmed false positives.

For both projects, we investigated the remaining issues manually in detail. There were several remaining false positives for various reasons: Due to the incompleteness of the procedure, e.g., missing further knowledge about functions calls, a path could not be identified as infeasible. Another reason is that we imposed a loop limit of 20 refinement iterations. Sometimes this limit was reached before a warning could be refuted. This happened 11 times in Wireshark, but only 3 times in Dovecot. As a side note, as discussed in [15], there are in general various reasons for false positives and often additional context information known to the developer is the key for refuting false positives. Moreover, it is worth noting that for path-insensitive checks (e.g., pattern matching) the number of false positives tends to be much lower or even zero.

Run-time Performance. The runtime results for the experiments are shown in Table 2. For the experiment we introduced timeouts both in Goanna as a whole as well as the SMT solver. For Goanna including the SMT path reduction loop an upper limit of 120s per file was set and in the SMT solver of 2s per solving. Moreover, we limited the maximum depth of SMT loops by 20. The timeouts, however, were only triggered very sporadically: Goanna timeouts occurred 12 times in Wireshark and once in Dovecot, which in both projects accounts for roughly 0.02% of all functions. Loop limits were reached similarly often and SMT timeouts occurred never in Wireshark and 5 times in Dovecot. In the remainder the analysis results are based on all non-timeout runs.

As shown in Table 2 the overall runtime for Wireshark was around two and a half hours, for Dovecot around $17\mathrm{min}$. In Wireshark for checks that can be improved through false path elimination around 15% of the runtime was spent in the SMT refinement loop. For the same objective the overhead in Dovecot was slightly higher with around 30%.

Interestingly, the vast majority of the overhead time is spent in the repeated model checking procedure rather than the SMT solving. Although the additional observers increase the state space in theory, the reachable state space will always be smaller than in the original model, since the observers constrain the set of reachable states. We have since then identified unnecessary overheads in our model checking procedure that should reduce the overall runtime in the future. However, given the value of a greatly reduced number of false positives, which can otherwise cost many engineering hours to identify, we believe that a run-time overhead of 15%–30% is already acceptable in practice; especially, if it equates to around $22\mathrm{min}$ in over one million lines of C/C++ code.

6 Related Work

Counter-example based path refinement with observers for static program analysis has been introduced by Fehnker et al. [14]. This work was based on using interval abstract interpretation to refute infeasible paths. While fast, it was limited to simple root causes for infeasible paths and much less precise than the SMT approach in this work. On the other hand, the application of predicate abstraction in conjunction with on-demand refinement has long been present in the CEGAR [6] approach and is used in many software model checkers such as SLAM [17] and BLAST [4,3]. This approach refines the whole model iteratively instead of eliminating sets of paths and using observers to learn from it. To an extent, a comparison of both approaches is still missing given their origin from different domains, namely static analysis and software mode checking.

The detection of infeasible paths and its use for program analysis has been explored by other authors, as well. Balakrishnan et al. [2] use this technique in the context of abstract interpretation. Delahaye et al. [11] present a technique how to generalize infeasible paths, but they have not investigated its use in static analysis. Yang et al. [26] propose the use of SMT solvers to remove infeasible paths by Dynamic Path Reduction. However, the work only addresses programs without pointers employing standard weakest precondition and it is not aimed at false-positive elimination. Harris et al. [16] describe a way to do program analysis by enumerating path programs. In contrast to our work they are not in a model-checking setting and their approach is not driven by counter-examples.

Finally, there are many examples of using SMT solvers in the realm of software model checking, e.g., as reasoning engine for bounded model checking [1,7].

7 Conclusions and Future Work

We have introduced a novel approach to reducing false positives in static program analysis. By treating static analysis as a syntactical model checking problem, we make static analysis amenable to an automata-based language refinement. Moreover, unlike traditional CEGAR approaches we create observer automata that exclude infeasible subpaths. The observers are computed based on a weakest precondition semantics using an SMT solver. We have shown that the approach works very well in practice and detects many relevant false positives.

Future work will further explore the limits of false-positive removal. We plan to investigate if more expensive SMT theories will lead to more false-positive removals or if, in fact, there are hardly any cases where this is necessary. Also, we will focus on further comparison with existing software model checking approaches and investigate if we can "out-source" some false-positive removal directly to a software model checker without much runtime penalty.

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. Int. J. Softw. Tools Techn. Transf. 11(1), 69–83 (2009)
2. Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Wei, O., Gupta, A.: SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 238–254. Springer, Heidelberg (2008)
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic Predicate Abstraction of C Programs. In: Proc. 2001 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2001), pp. 203–213. ACM (2001)
4. Ball, T., Rajamani, S.K.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
5. Burstall, R.: Some Techniques for Proving Correctness of Programs which Alter Data Structures. Mach. Intell. 7, 23–50 (1972)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In: Proc. 24th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2009), pp. 137–148. IEEE (2009)
8. Dams, D.R., Namjoshi, K.S.: Orion: High-Precision Methods for Static Error Analysis of C and C++ Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 138–160. Springer, Heidelberg (2006)
9. de la Banda, M., Stuckey, P., Wazny, J.: Finding All Minimal Unsatisfiable Subsets. In: 5th Int. ACM SIGPLAN Conf. Principles and Practice of Declarative Programming (PPDP 2003), pp. 32–43. ACM (2003)
10. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Delahaye, M., Botella, B., Gotlieb, A.: Explanation-Based Generalization of Infeasible Path. In: Proc. 3rd Int. Conf. Software Testing, Verification and Validation (ICST 2010), pp. 215–224. IEEE (2010)
12. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Trans. CAD Integ. Circ. Syst. 27(7), 1165–1178 (2008)
13. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model Checking Software at Compile Time. In: Proc. 1st Joint IEEE/IFIP Symp. Theoretical Aspects of Software Engineering (TASE 2007), pp. 45–56. IEEE (2007)

14. Fehnker, A., Huuck, R., Seefried, S.: Counterexample Guided Path Reduction for Static Program Analysis. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness. LNCS, vol. 5930, pp. 322–341. Springer, Heidelberg (2010)
15. Fehnker, A., Huuck, R., Seefried, S., Tapp, M.: Fade to Grey: Tuning Static Program Analysis. In: Proc. 3rd Int. Wsh. Harnessing Theories for Tool Support in Software (TTSS 2009), pp. 38–51. UNU-IIST (2009)
16. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program Analysis via Satisfiability Modulo Path Programs. In: Proc. 37th ACM SIGPLAN-SICACT Symp. Principles of Programming Languages (POPL 2010), pp. 71–82. ACM (2010)
17. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
18. Holzmann, G.J.: Static Source Code Checking for User-Defined Properties. In: Proc. 6th World Conf. Integrated Design and Process Technology (IDPT 2002). SDPS (2002)
19. Huuck, R., Fehnker, A., Seefried, S., Brauer, J.: Goanna: Syntactic Software Model Checking. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 216–221. Springer, Heidelberg (2008)
20. ISO/IEC. ISO/IEC 9899:2011 Information Technology – Programming Languages – C. ISO, Genève (2011)
21. Junker, M.: Using SMT Solvers for False Positive Elimination in Static Program Analysis. Master's thesis, Universität Augsburg (2010), `http://www4.in.tum.de/~junkerm/publications/thesis.pdf`
22. Liffiton, M.H., Sakallah, K.A.: On Finding All Minimally Unsatisfiable Subformulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 173–186. Springer, Heidelberg (2005)
23. Okun, V., Delaitre, A., Black, P.E. (eds.): Report on the Third Static Analysis Tool Exposition (SATE 2010). SP-500-283, U.S. Nat. Inst. Stand. Techn. (2011)
24. Schmidt, D.A., Steffen, B.: Program Analysis *as* Model Checking of Abstract Interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
25. Stump, A., Barrett, C., Dill, D., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: Proc. 16th Ann. IEEE Symp. Logic in Computer Science (LICS 2001), pp. 29–37. IEEE (2001)
26. Yang, Z., Al-Rawi, B., Sakallah, K., Huang, X., Smolka, S., Grosu, R.: Dynamic Path Reduction for Software Model Checking. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 322–336. Springer, Heidelberg (2009)