

Danger Invariants

Cristina David, Daniel Kroening, and Matt Lewis

University of Oxford

Abstract. Static analysers search for overapproximating proofs of safety commonly known as safety invariants. Fundamentally, such analysers summarise traces into sets of states, thus trading the ability to distinguish traces for computational tractability. Conversely, static bug finders (e.g. Bounded Model Checking) give evidence for the failure of an assertion in the form of a counterexample, which can be inspected by the user. However, static bug finders fail to scale when analysing programs with bugs that require many iterations of a loop as the computational effort grows exponentially with the depth of the bug. We propose a novel approach for finding bugs, which delivers the performance of abstract interpretation together with the concrete precision of BMC. To do this, we introduce the concept of *danger invariants* – the dual to safety invariants. Danger invariants summarise sets of traces that are guaranteed to reach an error state. This summarisation allows us to find deep bugs without false alarms and without explicitly unwinding loops. We present a second-order formulation of danger invariants and use the solver described in [1] to compute danger invariants for intricate programs taken from the literature.

Keywords: static bug finding, deep bugs, second-order logic, trace summarisation, program synthesis.

1 Introduction

Safety analysers search for proofs of safety commonly known as *safety invariants* by overapproximating the set of program states reached during all program executions. Fundamentally, they summarise traces into abstract states, thus trading the ability to distinguish traces for computational tractability [2]. Consequently, safety analysers may generate bug reports that do not correspond to actual errors in the code (i.e. *false alarms*). This is illustrated in Figure 1. False alarms are the primary barrier to the adoption of static analysis technology outside academia. Triage of true errors and false alarms is a tedious and difficult task, and there are reports that developers fare no better than coin tossing [3].

Conversely, static bug finders such as Bounded Model Checking (BMC) search for proofs that safety can be violated. Dually to safety proofs, we will call these *danger proofs*. Static bug finders have the attractive property that once an assertion fails, a counterexample trace is returned, which can be inspected by the user [4]. The counterexample is thus the proof that an assertion violation occurs. In order to construct such a danger proof, bounded model checkers

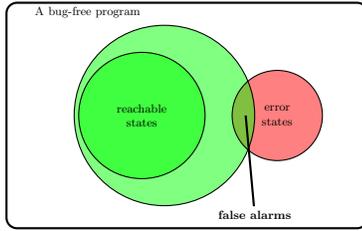


Fig. 1: Generation of false alarms

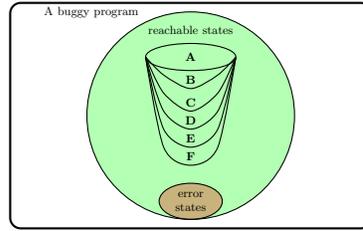


Fig. 2: Iterative unwinding of the transition relation

compute underapproximations of the reachable program states by progressively unwinding the transition relation. The downside of this approach is that static bug finders fail to scale when analysing programs with bugs that require many iterations of a loop. For illustration, Figure 2 depicts the successive unwinding of the transition relation, where progressively larger sets of reachable program states are labelled with letters from A to F. The computational effort required to discover an assertion violation (i.e. to obtain an intersection with the small ellipse labelled “error states”) typically grows exponentially with the depth of the bug.

Notably, the scalability problem is not limited to procedures that implement BMC. Approaches based on a combination of over- and underapproximations such as predicate abstraction [5] and lazy abstraction with interpolants (LAWI) [6] are not optimised for finding deep bugs either. The reason for this is that they can only detect counterexamples with deep loops after the repeated refutation of increasingly longer spurious counterexamples. The analyser first considers a potential error trace with one loop iteration, only to discover that this trace is infeasible. Consequently, the analyser increases the search depth, usually by considering one further loop iteration. This repeated unwinding suffers from the same exponential blow-up as BMC.

Danger proofs. In this paper we propose a novel representation of a danger proof based on trace summarisation. We propose to merge the two core concepts of safety analysers based on abstract interpretation and bug finders: trace summarisation (for scalability purposes) and counterexample generation (for precision). The intuition is that summarising traces is permissible as long as the summary is guaranteed to contain at least one feasible counterexample trace (i.e. a trace that starts in an initial state and reaches an error state).

The resulting summary is a dual of a safety invariant, which we refer to as a *danger invariant*. As opposed to safety invariants, danger invariants do *not* necessarily include all the reachable program states, but must contain at least one feasible execution trace. A danger invariant may encompass multiple paths through the program, but contains enough information to directly read off a concrete error trace. The danger invariant therefore amounts to a concise proof that such an error trace exists.

From a practical point of view, danger invariants will allow the development of bug finding techniques that do not require explicit loop unwinding. We empirically show that danger invariants improve the scalability of bug finding, enabling the detection of deep bugs. From a theoretical point of view, we propose a dual to over-approximating safety invariants, which are at the core of safety analysis. Over-approximating invariants have received an enormous quantity of research ever since the seminal work of Cousot and Cousot [2]. The equivalent of this body of work for invariants biased towards bug finding is missing.

While the main technique that we use in this paper to compute danger invariants is based on program synthesis (Section 3.1), given that the trace summarisation concept is common to both safety and danger invariants, we also investigate how methods used for safety invariant generation can be adapted for danger invariants (Section 5).

Contributions:

- To the best of our knowledge, we propose the first formulation of a *concise* proof of the existence of an error trace that allows trace summarisation without false alarms.
- We show that danger invariants improve the scalability of bug finding by using the second-order solver in [1] to infer danger invariants for programs with deep bugs.
- We investigate how techniques used for inferring safety invariants can be adapted for danger invariants.
- We generate danger invariants for a set of benchmarks taken from the literature.

2 From Counterexamples to Danger Invariants

We represent a program P as a transition system with state space X and transition relation $T \subseteq X \times X$. For a state $x \in X$ with $T(x, x')$, x' is said to be a successor of x under T . We denote initial states by I and error states by E .

Definition 1 (Execution Trace). *A program trace $\langle x_0 \dots x_n \rangle$ is a (potentially infinite, in which case $n = \omega$) sequence of states, such that any two successive states are related by the program’s transition relation T , i.e.*

$$\forall 0 \leq i < n. T(x_i, x_{i+1}).$$

Definition 2 (Counterexample). *A finite execution trace $\langle x_0 \dots x_n \rangle$ is a counterexample iff x_0 is an initial state, $x_0 \in I$, and x_n is an error state, $x_n \in E$.*

A counterexample is an instance of a danger proof: it provides evidence that an error state will be reached in some program execution. The question we try to answer in this paper is whether we can derive a more compact representation of a danger proof that does not require us to explicitly write down every intermediate state. Similarly to safety invariants, we obtain such a compact representation by

summarising traces. While this approach may involve overapproximation, the tricky part is retaining enough precision to ensure that a counterexample trace exists. In the rest of the section, we will explain our formulation by starting from a safety invariant and progressively adjusting it to show the existence of a counterexample. For this purpose, we will refer to loops of the form $L(G, T, I, A)$ shown in Figure 3, which we encode with predicates for the initial states: $I(x)$, guard: $G(x)$, body: $T(x, x')$ and assertion: $A(x)$.

```

assume (I);
while (G) T;
assert (A);

```

Fig. 3: The general form of a loop

Safety Invariants. Intuitively, a safety invariant is a set of states S that includes every state reachable via zero or more iterations of the loop, and which excludes all error states. More formally:

Definition 3 (Safety Invariant). *A predicate S is a safety invariant for the loop $L(I, G, T, A)$ iff it satisfies the following criteria:*

$$\forall x. I(x) \rightarrow S(x) \tag{1}$$

$$\forall x, x'. S(x) \wedge G(x) \wedge T(x, x') \rightarrow S(x') \tag{2}$$

$$\forall x. S(x) \wedge \neg G(x) \rightarrow A(x) \tag{3}$$

2.1 From Safety to Danger

It is well known that Definition 3 captures the notion of a safety invariant, and that if a predicate S exists that satisfies these three criteria, then the loop L is safe. We will now consider what the dual notion of a danger invariant might look like, and identify the criteria defining it.

Let us begin by considering what happens if we take the natural step of replacing criterion 3 with its complement: if we exit the loop in an S -state, we would like the assertion to fail. This gives us the following definition:

Definition 4 (Doomed Loop Head). *The head of the loop $L(I, G, T, A)$ is a doomed point [7] iff there exists a predicate S' satisfying:*

$$\forall x. I(x) \rightarrow S'(x) \tag{4}$$

$$\forall x, x'. S'(x) \wedge G(x) \wedge T(x, x') \rightarrow S'(x') \tag{5}$$

$$\forall x. S'(x) \wedge \neg G(x) \rightarrow \neg A(x) \tag{6}$$

The term “doomed program point” was introduced in [7] and denotes a program location that, whenever reached, will inevitably lead to an error regardless of the state in which it is reached. Definition 4 applies this notion to the head of the loop L : the predicate S' provides proof that the head of the loop L is a doomed point, so if such an S' exists then the loop will certainly fail. However, this definition is overly restrictive: in practice, for most unsafe programs such an S' does *not* exist.

For illustration, the program in Figure 4a is unsafe (any execution starting with $x > 10$ will lead to the assertion failing) but since there are *some* initial states that do not lead to an error (i.e. every state with $x \leq 10$) there are no doomed points. Thus, we weaken Definition 4 by introducing *doomed program states*: we weaken criterion 1 to say that there must be *some* initial state leading to an error rather than *every* initial state. This gives us the following:

Definition 5 (Doomed Program State). *There is trace containing an error state, starting from the head of the loop $L(I, G, T, A)$ if a predicate S'' exists satisfying:*

$$\exists x_0. I(x_0) \rightarrow S''(x_0) \tag{7}$$

$$\forall x, x'. S''(x) \wedge G(x) \wedge T(x, x') \rightarrow S''(x') \tag{8}$$

$$\forall x. S''(x) \wedge \neg G(x) \rightarrow \neg A(x) \tag{9}$$

<pre>x = *; while (x < 10) { x++; } assert (x == 10);</pre>	<pre>x = 0; while (x < 10) { if (*) break; x++; } assert (x == 10);</pre>	<pre>x = 0; y = 0; while (x < 10) { y++; } assert (x < 10);</pre>
--	--	---

- | | | |
|--|--|--|
| <p>(a) An unsafe loop with no doomed program points.</p> | <p>(b) An unsafe loop with no doomed states.</p> | <p>(c) A safe program with no finite error traces.</p> |
|--|--|--|

Fig. 4: Illustrative programs – * means nondeterministic choice.

Definition 5 weakens the notion of a doomed location. Rather than requiring every trace including the doomed location to reach an error, we only require that there is some doomed initial state, i.e. every trace including that *state* will reach an error. However, this is still too strong a condition! Figure 4b shows an unsafe program which has no doomed states – every state in the loop has some successor that leads to a state in which the assertion holds (i.e. every trace in which the break is never executed). To work around this, we can weaken criterion 2 to say that each state must have *some* successor leading to an error.

Definition 6 (Partial Danger). *There is a trace containing an error state, starting from the head of the loop $L(I, G, T, A)$ if a predicate S''' exists satisfying:*

$$\exists x_0. I(x_0) \rightarrow S'''(x_0) \quad (10)$$

$$\forall x. S'''(x) \wedge G(x) \rightarrow \exists x'. \wedge T(x, x') \wedge S'''(x') \quad (11)$$

$$\forall x. S'''(x) \wedge \neg G(x) \rightarrow \neg A(x) \quad (12)$$

We are very nearly done: Definition 6 captures that there is some trace containing an error state starting from some initial state. However, our definition of an execution trace (Definition 1) includes infinite traces. Thus, the trace containing the error may be infinite and the error state will not be reachable at all. For example, consider Figure 4c. A possible partial danger invariant is ‘true’, which meets all of the criteria 10, 11 and 12. However, the program is in fact safe – it contains no terminating traces and so the assertion is never even reached. Definition 6 captures *partial danger*, which disregards the termination behaviour of the program. Instead, what we really want is *total danger*, which will guarantee that there is some finite trace culminating in an error. To ensure that the error traces are finite, we will introduce a *ranking function*, which will serve as a proof of termination. Below we recall the definition of a ranking function:

Definition 7 (Ranking function). *A function $R : X \rightarrow Y$ is a ranking function for the transition relation T if Y is a well-founded set with order $>$ and R is injective and monotonically decreasing with respect to T . That is to say:*

$$\forall x, x' \in X. T(x, x') \Rightarrow R(x) > R(x')$$

This is the final piece we need to define danger invariants:

Definition 8 (Danger Invariant). *A pair $\langle D, R \rangle$ of a predicate and a ranking function is a danger invariant for the loop $L(I, G, T, A)$ iff it satisfies the following criteria:*

$$\exists x_0. I(x_0) \wedge D(x_0) \quad (13)$$

$$\forall x. D(x) \wedge G(x) \rightarrow R(x) > 0 \wedge \exists x'. T(x, x') \wedge D(x') \wedge R(x') < R(x) \quad (14)$$

$$\forall x. D(x) \wedge \neg G(x) \rightarrow \neg A(x) \quad (15)$$

Theorem 1 (Danger Invariants Prove Bugs). *The loop $L(I, G, T, A)$ is unsafe iff there exists a danger invariant satisfying the criteria in Definition 8. In other words, the existence of a danger invariant is a necessary and sufficient condition for the reachability of a bug.*

3 Second-Order Formulation of Danger Invariants

The problem of program verification can be reduced to the problem of finding solutions to a second-order constraint [8, 9]. In order to give a second-order formulation of a danger invariant, we will use a fragment of second-order logic decidable over finite domains that we defined in [1], and to whose satisfiability problem we refer as Second-Order SAT. Next, we recall this fragment:

Definition 9 (Second-Order SAT).

$$\exists S_1 \dots S_m. Q_1 x_1 \dots Q_n x_n. \sigma$$

Where the S_i range over predicates, the Q_i are either \exists or \forall , x_i range over Boolean values and σ is a quantifier-free propositional formula that may refer to both the first-order variables x_i and the second-order variables S_i .

Definition 10 (Danger Invariant Formula [DI]).

$$\begin{aligned} \exists D, R, x_0. \forall x. \exists x'. I(x_0) \wedge D(x_0) \wedge \\ D(x) \wedge G(x) \rightarrow T(x, x') \wedge D(x') \wedge \\ R(x) > 0 \wedge R(x) > R(x') \wedge \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x) \end{aligned}$$

Definition 11 (Skolemized Danger Invariant Formula [SDI]).

$$\begin{aligned} \exists D, R, N, x_0. \forall x. I(x_0 \wedge D(x_0)) \wedge \\ D(x) \wedge G(x) \rightarrow R(x) > 0 \wedge T(x, N(x)) \wedge D(N(x)) \wedge R(x) > R(N(x)) \\ D(x) \wedge \neg G(x) \rightarrow \neg A(x) \end{aligned}$$

Fig. 5: Existence of a danger invariant as second-order SAT

Our first second-order formulation of a danger invariant is captured in the second-order SAT formula [DI] of Definition 10. In this definition, in order to specify that from each D -state we can reach another by iterating the loop once, we require quantifier alternation over the first order variables. However, the solver from [1] that we want to use requires eliminating the extra level of quantifier alternation (the inner existential quantifier) by using Skolem functions.

If the transition relation T is deterministic, then we do not need the quantifier alternation, since each x has exactly one successor x' . Thus, we can just replace the inner $\exists x'$ in the formula [DI] by $\forall x'$. However, if T is non-deterministic, we must find a Skolem function N which resolves the non-determinism by telling us exactly which successor is to be chosen on each iteration of the loop. This is shown in the formula [SDI] of Definition 11.

3.1 Danger Invariants Generation

In this section, we discuss how to solve the constraints generated for danger invariants. In [1], we show that Second-Order SAT is polynomial-time reducible to finite synthesis, and finite-state program synthesis is NEXPTIME-complete.

Next, we provide a short description of the program synthesis algorithm and, for more details, we direct the reader to [1]. Our algorithm is sound and complete for the finite-state synthesis decision problem. In the case that a specification is satisfiable, our algorithm produces a *minimal* satisfying program. We use Counterexample Guided Inductive Synthesis (CEGIS) [10, 11] to find a program satisfying our specification.

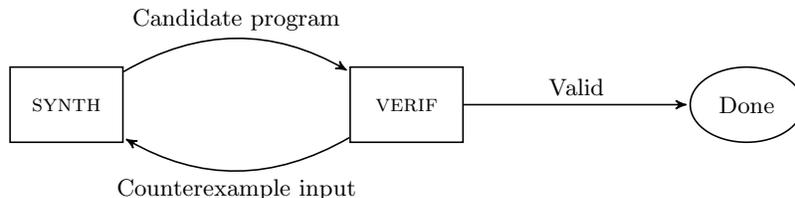


Fig. 6: Abstract synthesis refinement loop

As illustrated in Figure 6, the algorithm is divided into two procedures: SYNTH and VERIF, which interact via a finite set of test vectors INPUTS. By using explicit proof search, symbolic bounded model checking and genetic programming with incremental evolution [12, 13], the SYNTH procedure tries to find an existential witness P that satisfies the partial specification:

$$\exists P. \forall x \in \text{INPUTS}. \sigma(x, P)$$

If SYNTH succeeds in finding a witness P , this witness is a candidate solution to the full synthesis formula. We pass this candidate solution to VERIF which determines whether it does satisfy the specification on all inputs by checking satisfiability of the verification formula:

$$\exists x. \neg \sigma(x, P)$$

If this formula is unsatisfiable, the candidate solution is in fact a solution to the synthesis formula and so the algorithm terminates. Otherwise, the witness x is an input on which the candidate solution fails to meet the specification. This witness x is added to the INPUTS set and the loop iterates again.

3.2 Generalised Safety-Danger Formula

The decision procedure introduced in [1] and briefly described above relies on a small-model argument to determine that a formula is unsatisfiable and, in practice, it is usually unable to prove unsatisfiability. Therefore we would like to only provide satisfiable formulae whenever possible. Since the program we are analysing is either safe or unsafe, and assuming that a proof is expressible in our logic, a program either accepts a safety invariant or a danger invariant.

We model this as a disjunction in the formula **[GS]** of Definition 12. **[GS]** is a theorem of second-order logic, and our decision procedure will always be able to find witnesses S, D, N, R, y_0 demonstrating its truth, provided such a witness is expressible in our logic. The synthesised predicate S is a purported safety invariant and the D, N, R, y_0 constitute a purported danger invariant. If S is really a safety invariant, the program is safe, otherwise D, R (with witnesses to the existence of an error trace with Skolem function N and initial state y_0) will be a danger invariant and the program is unsafe. Exactly one of these proofs will be valid, i.e. either S will satisfy the criteria for a safety invariant, or D, N, R, y_0 will satisfy the criteria for a danger invariant. We can simply check both cases and discard whichever “proof” is incorrect.

Definition 12 (Generalised Safety Formula [GS]).

$$\exists S, D, N, R, y_0. \forall x, x', y. \left(\begin{array}{l} I(x) \rightarrow S(x) \wedge \\ S(x) \wedge G(x) \wedge T(x, x') \rightarrow S(x') \wedge \\ S(x) \wedge \neg G(x) \rightarrow A(x) \end{array} \right) \vee \left(\begin{array}{l} I(y_0) \wedge D(y_0) \wedge \\ D(y) \wedge G(y) \rightarrow R(y) > 0 \wedge T(y, N(y)) \wedge D(N(y)) \\ \wedge R(y) > R(N(y)) \wedge \\ D(y) \wedge \neg G(y) \rightarrow \neg A(y) \end{array} \right)$$

Fig. 7: General second-order SAT formula characterising safety

4 Illustrative Examples

To illustrate the use of danger invariants for compactly representing danger proofs and finding deep bugs, as well as the shortcomings of the existing bug finding techniques, we consider the programs in Figure 8, where (a), (b), (c), (d) contain deep bugs, whereas (e) models a buffer overflow.

For program (a), any execution trace violates the assertion unless the non-deterministic choice (denoted by “*”) is such that y is incremented exactly 999999 times out of the 1000000 iterations of the loop. In order to analyse this program bounded model checkers have to completely unwind the loop. The resulting SAT instance is very large, so large in fact that solving it will almost certainly take far too long to be practical. Hybrid approaches such as predicate abstraction and LAWI will have to progressively unwind the loop in order to refute spurious counterexamples of increasing length. Again, this is most likely to take too long to be practical.

In our case, in order to prove that the assertion can be violated, we need to find a danger invariant. This means that we need an approximation of the set of states reachable during the program's execution that must include a feasible counterexample trace. Of course there may be several such invariants. One possibility for this example is $D(x, y) = x < y$ and ranking function $R(x, y) = -x$. D holds in the initial state where $x = 0$ and $y = 1$, and it is inductive with respect to the loop's body if the nondeterministic choice is given by the Skolem function $N_y(x, y) = y + 1$. That is:

$$\forall x, y, x'. x < y \rightarrow x' = x + 1 \wedge x' < N_y(x, y)$$

Program (b) is a version of (a) with additional nondeterminism. Now, in each loop iteration, x may or may not be incremented. This modification substantially increases the number of reachable program states, as well as introducing a potential non-terminating behaviour as x may not reach 1000000. As a result, bounded model checkers will loop forever trying to generate the SAT instance corresponding to the unwound loop. Similarly, predicate abstraction and LAWI based approaches are even less likely to be practical than for version (a) of the program due to the increase in the number of reachable states. We synthesise the same D as for program (a) as it still contains a feasible counterexample trace regardless of the additional nondeterminism. Here, termination depends on the Skolem function giving us the successor of x . Thus, we can have $R(x, y) = -x$ and $N_x(x, y) = x + 1$.

Program (c) is similar to (a), with the exception that the assertion is now negated. This example is more intricate as the danger invariant needs to capture the evolution of x and y from the the initial state where they are not equal to a final state where there are (and hence they cause the assertion to fail). One such invariant is $D(x, y) = y == (x < 1 ? 1 : x)$ and $R(x, y) = -x$. Essentially, this invariant says that y must not be incremented for the first iteration of the loop (until x reaches the value 1), and from that point, for the rest of the iterations, y gets always incremented such that $x == y$. For this case, D is a compact and elegant representation of exactly one feasible counterexample trace. The witness Skolem function that we get is $N_y(x, y) = (x < 1 ? y : y + 1)$.

Program (d) is taken from [14], and it is meant to illustrate the difficulty faced by tools based on predicate abstraction and abstraction refinement with deterministic loops with a fixed execution count as many iterations of the iterative refinement algorithm correspond to spurious executions of the loop body. The assertion fails, but proving this requires 1000000 iterations of the refinement loop, resulting in the introduction of the predicates $(i == 0), (i == 1), \dots, (i == 1000000)$ one by one. For us, some of the possible danger invariants are $D(i, c, a) = a \leq -10$, $D(i, c, a) = a == 0$, $D(i, c, a) = a \leq 0$ and $R(i, c, a) = true$.

Program (e) models a check for a buffer overflow. The buffer overflow does happen whenever x is big enough such that the computation of $x * 4$ overflows. We find the danger invariant $D(x, i, len) = (i == 0 \wedge len == 0 \wedge x \neq 0)$ and ranking function $R(x, i, len) = true$. This basically says that the computation of

len overflowed resulting in $len = 0$ (while $x \neq 0$). Consequently, the loop is not taken such that i stays 0 and the ranking function is *true*. Note that this example has the assertion inside the loop, and required some trivial preprocessing in order to move it outside the loop.

```
x = 0; y = 1;
while (x < 1000000) {
    x++;
    if (*) y++;
}
```

```
assert ( x == y ) ;
```

(a)

```
x = 0; y = 1;
while (x < 1000000) {
    if (*) x++;
    if (*) y++;
}
```

```
assert ( x == y ) ;
```

(b)

```
x = 0; y = 1;
while (x < 1000000) {
    x++;
    if (*) y++;
}
```

```
assert ( x != y ) ;
```

(c)

```
void foo(int a)
{
    int i, c;
    i = 0;
    c = 0;
    while (i < 1000000) {
        c=c+i;
        i=i+1;
    }
    assert (a > 0);
}
```

(d) Adapted from [14]

```
int main(void) {
    unsigned int x, i, len;

    len = x * 4;

    for (i = 0; i < x; i++) {
        assert(i * 4 < len)
    }
}
```

(e) Checking for a buffer overflow caused by an integer overflow.

Fig. 8: Motivational examples.

5 Danger Invariants in Relation to Other Formalisms

In this section, we relate danger invariants to other existing formalisms. For this purpose, we start by providing a characterisation of danger invariants as a fixed point computation. First, we define the set of program executions starting in an initial state, E_{fwd} , and the set of program executions ending in an error

state, E_{bck} . The set of program execution traces starting in an initial state (real executions) is

$$T_{fwd} = \{\langle x_0 \dots x_n \rangle \mid \forall i. T(x_i, x_{i+1}) \wedge x_0 \in I\} = \text{lfp}(F_{fwd})$$

where F_{fwd} constructs execution traces by taking a transition forward:

$$F_{fwd} = \lambda S. \{\langle x \rangle \mid x \in I\} \cup \{\langle x_0 \dots x_n, x_{n+1} \rangle \mid \langle x_0 \dots x_n \rangle \in S \wedge T(x_n, x_{n+1})\}$$

The set of program execution traces ending in an error state is

$$T_{bck} = \{\langle x_0 \dots x_n \rangle \mid \forall i. T(x_i, x_{i+1}) \wedge x_n \in E\} = \text{lfp}(F_{bck})$$

where F_{bck} constructs execution traces by taking a transition backward:

$$F_{bck} = \lambda S. \{\langle x \rangle \mid x \in E\} \cup \{\langle x_{-1}, x_0 \dots x_n \rangle \mid \langle x_0 \dots x_n \rangle \in S \wedge T(x_{-1}, x_0)\}$$

Similar to [15], we can now express the set of execution traces starting in an initial state and ending in an error state as $Err = \text{lfp}(F_{fwd}) \cap \text{lfp}(F_{bck})$. Then, a danger invariant is an approximation of $\text{lfp}(F_{fwd})$ (either $D \supseteq \text{lfp}(F_{fwd})$ or $D \subseteq \text{lfp}(F_{fwd})$), such that it contains at least one error trace $D \cap Err \neq \emptyset$.

5.1 Abstract Interpretation

Given that direct computation of the set of possible execution traces of a program is most of the times infeasible, an abstract interpretation based analysis conservatively computes at each program point a set of abstract states representing an overapproximation of the possible concrete program states [2]. The analysis assigns to each program location an abstract value from an abstract domain. The concretisation mapping γ is defined such that for every program location l produces an abstract state $x^\#$, such that $\gamma(x^\#)$ contains all the concrete states reachable at location l .

The abstract forward interpreter $F_{fwd}^\#$ is inherently overapproximating, $\text{lfp}(F_{fwd}) \subseteq \gamma(\text{lfp}^\#(F_{fwd}^\#))$. This means that, whenever an error is signalled at program location l , the alarm may be spurious. In order to decide whether an alarm is genuine, abstract interpretation based techniques require backward analysis starting from the error state. Various solutions have been already proposed in the abstract interpretation literature [16, 17, 15]. Next, we see which of these can be used to compute danger invariants.

For illustration purposes, we will use the program in Figure 9 (adapted from [16]) as a running example throughout this section. The program takes an input variable y bounded between 100 and 200 and iteratively decreases it by 2. The program is erroneous as the assertion $y == 0$ is violated whenever the initial value of y is odd. The abstract states $x_0^\#$ to $x_4^\#$ inferred by a forward analysis based on the interval domain are listed next: $x_0^\# = [100, 200]$, $x_1^\# = [-1, 200]$, $x_2^\# = [1, 200]$, $x_3^\# = [-1, 198]$, $x_4^\# = [-1, 0]$.

Since $x_4^\#$ violates the assertion $y == 0$, in order to check whether it is genuine, the negation of the assertion $y < 0 \wedge y > 0$ must be propagated backwards.

Given that the result of the forward analysis is a sound overapproximation, the negation of the assertion can be intersected with x_4^\sharp resulting in $y = -1$, which is then propagated backwards. If this propagation results in the set of states being empty at any program location, then the error is a false alarm. Otherwise, it is genuine. The main challenge is the presence of loops. Given a state after a loop, it is non-trivial to infer a state that is valid prior to entering the loop. In particular, it is necessary to assess how often the loop body needs to be executed to reach the exit state.

In [16] Brauer and Simon use an overapproximating affine analysis to estimate the number of loop iterations. Thus, they are able to obtain $y = 2n - 1 \wedge n \in [1, 63]$ at location l_1 (corresponding to x_1^\sharp in the CFG). In order to obtain a danger invariant from this, we just need to add the error state $y = -1$. Thus, we obtain $D(y) = (y == 2n - 1 \wedge n \in [0, 63])$, which is guaranteed to contain a concrete counterexample. In [17], Erez performs a bounded search for backward traces up to a given depth. For the given example, the first counterexample found is: $\langle y = -1, y = 1, \dots, y = 101 \rangle$. Thus, a corresponding danger invariant is $D(y) = (y == -1 \vee y == 1 \vee \dots \vee y == 101)$, or $D(y) = (y == 2n - 1 \wedge n \in [0, 51])$. While the results of both these techniques can be immediately used to compute danger invariants, this is not true for other works on eliminating false alarms in abstract interpretation based techniques such as [15], where Rival computes an overapproximation of Err by using in his backward analysis the same domains as in forward analysis. Thus, the result is not a danger invariant and cannot ensure the existence of a true error.

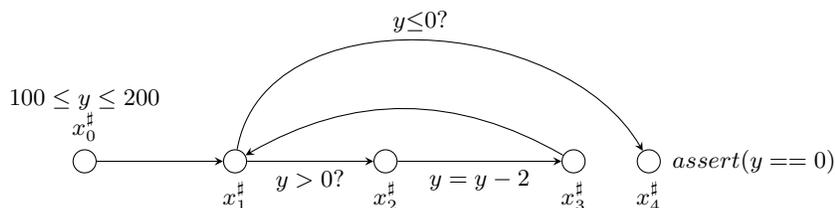


Fig. 9: The CFG of an erroneous program: the labels on the vertices denote the corresponding abstract program states and the arcs correspond to instructions in the program.

5.2 Bounded Model Checking

For a loop $L(I, G, T, A)$, a bounded model checker progressively unwinds the transition relation up to a depth k . As such:

$$T_{fwd}^k = \text{lfp}(F_{fwd}^k)$$

where F_{fwd}^k constructs execution traces by taking a transition forward:

$$F_{fwd}^k = \lambda S. \{ \langle x \rangle \mid x \in I \} \cup \{ \langle x_0 \dots x_n, x_{n+1} \rangle \mid \langle x_0 \dots x_n \rangle \in S \wedge T(x_n, x_{n+1}) \wedge n < k \}$$

Thus, BMC finds counterexample traces of the form $\langle x_0, \dots, x_n \rangle$, where $x_0 \in I$, $x_n \in E$ and $n \leq k$. Such a counterexample directly corresponds to the danger invariant $D(x) = \bigvee_{i=0, n} x_i$. For the running example, we have used CBMC [18] and obtained the counterexample trace $\langle y = 101, y = 99, \dots, y = -1 \rangle$, which corresponds to the danger invariant $D(y) = (y == 101 \vee y == 99 \vee \dots \vee y == -1)$.

5.3 Linear Invariants

There is a lot of work on the generation of linear invariants of the form $c_1x_1 + \dots + c_nx_n + d \leq 0$ [19, 20]. The main idea behind these techniques is to treat the coefficients c_1, \dots, c_n, d as unknowns and generate constraints on them such that any solution corresponds to a safety invariant. In [20], Colon et al. present a method based on Farkas' Lemma, which synthesises linear invariants by extracting non-linear constraints on the coefficients of a target invariant from a program. In a different work, Sharma and Aiken use randomised search to find the coefficients [20]. It would be interesting to investigate how these methods can be adapted for generating constraints on the coefficients c_1, \dots, c_n, d such that solutions correspond to linear danger invariants.

6 Experimental Results

To evaluate our algorithm, we implemented the DANGERZONE tool, which generates a danger specification from a C program and calls the second-order SAT solver discussed in [1] to obtain a proof. We ran the resulting prover on 20 buggy programs including the running examples in the paper, some examples from the literature and some from SV-COMP'15 [21]. Our benchmarks do not make use of arrays or recursion. We do not have arrays in our logic and we had not implemented recursion in our frontend (although the latter can be syntactically rewritten to our input format).

For each benchmark we infer a danger invariant, a ranking function, an initial state and Skolem functions witnessing the nondeterminism. To provide a comparison point, we also ran CBMC [18] on the same benchmarks. For CBMC, we manually provided sufficient unwinding limits for each program. Each tool was given a time limit of 1800s, and was run on an unloaded 4-core 3.30 GHz i5-2500k with 8 GB of RAM. The results of these experiments are given in Figure 10.

On programs with shallow bugs (i.e. bugs that can be reached after a small number of loop iterations), CBMC is much faster than DANGERZONE. However, DANGERZONE performs much better on programs with deep bugs providing empirical evidence that danger invariants improve the scalability of static bug finding. We feel that the performance difference for shallow bugs is inherent

to the difference in the two approaches – our solver is more general (and less engineered) than CBMC, in that it provides a complete proof system for both danger and safety. The two examples on which DANGERZONE times out have an intricate control flow structure which requires specific constants to be inferred in the danger invariant – a case for which DANGERZONE is not optimised. Notably, only 6 of the benchmarks (marked with * in the table) contain doomed loop heads.

Benchmark	Deep Bugs	Shallow Bugs	CBMC	DANGERZONE
loop1.c*	✓	-	575 s	4.32 s
loop2.c	✓	-	TO	6.64 s
loop3.c [Fig 8c]	✓	-	TO	76.58 s
loop4.c [Fig 8e]	-	✓	0.082 s	8.82 s
loop5.c [Fig 8d]	✓	-	TO	8.89 s
loop6.c [14]	-	✓	0.084 s	6.51 s
loop7.c*	✓	-	TO	8.42 s
loop8.c*	✓	-	TO	5.09 s
loop9.c*	✓	-	TO	9.68 s
loop10.c	✓	-	TO	6.59 s
loop11.c [Fig 9]	-	✓	0.171 s	5.54 s
loop12.c*	-	✓	0.081 s	12.09 s
loop13.c	-	✓	0.256 s	7.22 s
loop14.c	-	✓	0.286 s	TO
loop15.c	-	✓	2.111 s	TO
loop16.c*	-	✓	0.081 s	8.20 s
loop17.c [Fig 8a]	✓	-	TO	6.30 s
loop18.c	✓	-	TO	8.59 s
loop19.c [Fig 8b]	✓	-	TO	7.40 s
loop20.c [Fig 4a]	-	✓	0.082 s	6.97 s

Key: TO = time-out

Fig. 10: Experimental results

7 Conclusions

In this paper, we introduced the concept of danger invariants – the dual to safety invariants. Danger invariants summarise sets of traces that are guaranteed to reach an error state. This summarisation allows us to find deep bugs without false alarms and without explicitly unwinding loops. This new concept promises to deliver the performance of abstract interpretation together with the concrete precision of BMC. We presented a second-order formulation of danger invariants and used the solver described in [1] to compute danger invariants for a set of benchmarks.

References

1. David, C., Kroening, D., Lewis, M.: Second-order propositional satisfiability. *CoRR* (2014)
2. 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
3. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: *Programming Language Design and Implementation (PLDI)*. (2012) 181–192
4. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Formal Methods in System Design* **19**(1) (2001) 7–34
5. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* (1994) 1512–1542
6. McMillan, K.L.: Lazy abstraction with interpolants. In: *Computer Aided Verification (CAV)*. (2006) 123–136
7. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It’s doomed; we can prove it. In: *FM 2009: Formal Methods*. Springer (2009) 338–353
8. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: *Programming Language Design and Implementation (PLDI)*. (2012) 405–416
9. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: *Programming Language Design and Implementation (PLDI)*. (2008) 281–292
10. Solar Lezama, A.: Program Synthesis By Sketching. PhD thesis, EECS Department, University of California, Berkeley (Dec 2008)
11. Solar-Lezama, A.: Program sketching. *STTT* **15**(5-6) (2013) 475–495
12. Langdon, W.B., Poli, R.: *Foundations of Genetic Programming*. Springer (2002)
13. Brameier, M., Banzhaf, W.: *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer (2007)
14. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: *Foundations of Software Engineering (FSE)*. (2006) 117–127
15. Rival, X.: Understanding the origin of alarms in Astrée. In: *Static Analysis (SAS)*, Springer (2005) 303–319
16. Brauer, J., Simon, A.: Inferring definite counterexamples through underapproximation. In: *NASA Formal Methods (NFM)*. (2012) 54–69
17. Erez, G.: Generating concrete counterexamples for sound abstract interpretation. Master’s thesis, Tel-Aviv University (2004)
18. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: *TACAS*. Springer (2004) 168–176
19. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: *Computer Aided Verification (CAV)*. (2003) 420–432
20. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: *CAV*. (2014) 88–105
21. SV-COMP 2015: <http://sv-comp.sosy-lab.org/2015/>.