# Under-Approximating Loops in C Programs for Fast Counterexample Detection*

Daniel Kroening[1], Matt Lewis[1], and Georg Weissenbacher[2]**

[1] Oxford University
[2] Vienna University of Technology, Austria

**Abstract.** Many software model checkers only detect counterexamples with deep loops after exploring numerous spurious and increasingly longer counterexamples. We propose a technique that aims at eliminating this weakness by constructing auxiliary paths that represent the effect of a range of loop iterations. Unlike acceleration, which captures the exact effect of arbitrarily many loop iterations, these auxiliary paths may under-approximate the behaviour of the loops. In return, the approximation is sound with respect to the bit-vector semantics of programs.
Our approach supports arbitrary conditions and assignments to arrays in the loop body, but may as a result introduce quantified conditionals. To reduce the resulting performance penalty, we present two quantifier elimination techniques specially geared towards our application.
Loop under-approximation can be combined with a broad range of verification techniques. We paired our techniques with lazy abstraction and bounded model checking, and evaluated the resulting tool on a number of buffer overflow benchmarks, demonstrating its ability to efficiently detect deep counterexamples in C programs that manipulate arrays.

## 1 Introduction

The generation of *diagnostic counterexamples* is a key feature of model checking. Counterexamples serve as witness for the refutation of a property, and are an invaluable aid to the engineer for understanding and repairing the fault.

Counterexamples are particularly important in software model checking, as bugs in software frequently require thousands of transitions to be executed, and are thus difficult to reproduce without the help of an explicit error trace. Existing software model checkers, however, fail to scale when analysing programs with bugs that involve many iterations of a loop. The primary reason for the inability of many existing tools to discover such "deep" bugs is that exploration is performed in a breadth-first fashion: the detection of an unsafe execution traversing a loop involves the repeated refutation of increasingly longer spurious

* Supported by the Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/H017585/1, the EU FP7 STREP PINCETTE, the ARTEMIS VeTeSS project, and ERC project 280053.
** Funded by the Vienna Science and Technology Fund (WWTF) through project VRG11-005 and the Austrian Science Fund (FWF) through RiSE (S11403-N23).

counterexamples. The analyser first considers a potential error trace with one loop iteration, only to discover that this trace is infeasible. As consequence, the analyser will increase the search depth, usually by considering one further loop iteration. In practice, the computational effort required to discover an assertion violation thus grows exponentially with the depth of the bug.

Notably, the problem is not limited to procedures based on abstraction, such as predicate abstraction or abstraction with interpolants. Bounded Model Checking (BMC) is optimised for discovering bugs up to a given depth $k$, but the computational cost grows exponentially in $k$.

The contribution of this paper is a new technique that enables scalable detection of deep bugs. We transform the program by adding a new, auxiliary path to loops that summarises the effect of a parametric number of iterations of the loop. Similar to acceleration, which captures the exact effect of arbitrarily many iterations of an integer relation by computing its reflexive transitive closure in one step [3, 6, 9], we construct a summary of the behaviour of the loop. By symbolically bounding the number of iterations, we obtain an *under-approximation* which is sound with respect to the bit-vector semantics of programs. Thus, we avoid false alarms that might be triggered by modeling variables as integers.

In contrast to related work, our technique supports assignments to arrays and arbitrary conditional branching by computing quantified conditionals. As the computational cost of analysing programs with quantifiers is high, we introduce two novel techniques for summarising certain conditionals without quantifiers. The key insight is that many conditionals in programs (e.g., loop exit conditions such as $\texttt{i} \leq 100$ or even $\texttt{i} \neq 100$) exhibit a certain monotonicity property that allows us to drop quantifiers.

Our approximation can be combined soundly with a broad range of verification engines, including predicate abstraction, lazy abstraction with interpolation [16], and bounded software model checking [4]. To demonstrate this versatility, we combined our technique with lazy abstraction and the CBMC [4] model checker. We evaluated the resulting tool on a large suite of benchmarks known to contain deep paths, demonstrating our ability to efficiently detect deep counterexamples in C programs that manipulate arrays.

## 2 Outline

### 2.1 Notation and Preliminaries

We restrict our presentation to a simple imperative language comprising assignments, assumptions, and assertions. A program is a control flow graph $\langle V, E, \lambda \rangle$, where $V$ and $E$ are sets of vertices and edges, respectively, and $\lambda$ is a labelling function mapping vertices to statements. Procedure calls are in-lined and omitted in our presentation. The behaviour of a program is defined by the paths in the control flow graph (CFG). A path $\pi$ of length $m$ is a sequence of contiguous edges $e_1\, e_2 \ldots e_m$ ($e_i \in E$, $1 \leq i \leq m$). Abusing our notation, we use the corresponding sequence of statements $\lambda(e_1); \lambda(e_1); \ldots \lambda(e_m)$ to represent paths (where

| **Path** $\pi$ | **Strongest Postcondition** $sp(\pi, P)$ | **Weakest Liberal Precondition** $wlp(\pi, Q)$ |
|---|---|---|
| $\varepsilon$ / `skip` | $P$ | $Q$ |
| `x:=`$e$ | $\exists \grave{\mathtt{x}} \,.\, (\mathtt{x} = e[\mathtt{x}/\grave{\mathtt{x}}]) \wedge P[\mathtt{x}/\grave{\mathtt{x}}]$ | $Q[\mathtt{x}/e]$ |
| $[R]$ | $P \wedge R$ | $R \Rightarrow Q$ |
| `assert(`$R$`)` | $P \wedge R$ | $R \Rightarrow Q$ |
| $\pi_1$ `;` $\pi_2$ | $sp(\pi_2, sp(\pi_1, P))$ | $wlp(\pi_1, wlp(\pi_2, Q))$ |
| $\pi_1 \sqcap \pi_2$ | $sp(\pi_1, P) \vee sp(\pi_2, P)$ | $wlp(\pi_1, Q) \wedge wlp(\pi_2, Q)$ |

Table 1: Predicate transformers for simple program statements and paths. $Q[\mathtt{x}/e]$ denotes that all free occurrences of $\mathtt{x}$ in $Q$ are replaced with the expression $e$.

`;` denotes the non-commutative path concatenation operator). We use $\varepsilon$ to denote the path of length 0 and inductively define $\pi^n$ as $\pi^0 = \varepsilon$ and $\pi^{n+1} = \pi^n; \pi$ (for $n \geq 0$). In accordance with [17], $\pi_1 \sqcap \pi_2$ represents the non-deterministic choice between two paths, i.e., $\overset{\pi_1}{\underset{\pi_2}{\circ\!\!\rightrightarrows\!\!\circ}}$. The commutative operator $\sqcap$ is extended to sets of paths in the usual manner.

We use first-order logic (defined as usual) with background theories commonly used in software verification (such as arithmetic, bit-vectors, arrays and uninterpreted functions) to represent program expressions and predicates. $\mathsf{T}$ ($\mathsf{F}$) represents the predicate that is always true (false). We use $*$ to indicate non-deterministic values. The semantics for statements and paths is determined by the predicate transformers in Table 1 (see [17]). A Hoare triple $\{P\}\ \pi\ \{Q\}$ comprises a pre-condition $P$, a path $\pi$, and a post-condition $Q$ such that $sp(\pi, P)$ implies $Q$. Given a set $\mathtt{X} = \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ of $k$ variables, we introduce corresponding sets $\grave{\mathtt{X}} = \{\grave{\mathtt{x}}_1, \ldots, \grave{\mathtt{x}}_k\}$ and $\mathtt{X}' = \{\mathtt{x}'_1, \ldots, \mathtt{x}'_k\}$ of *primed* variables to refer to variables in prior and subsequent time-frames, respectively (where the term *time-frame* refers to an instance of $\pi$ in $\pi^n$). We use $\mathtt{X}^{\langle i \rangle} = \{\mathtt{x}_1^{\langle i \rangle}, \ldots, \mathtt{x}_k^{\langle i \rangle}\}$ to refer to the variables in a specific time-frame $i$. The transition relation of $\pi$ is the predicate $\neg wlp(\pi, \bigvee_{i=1}^{k} \mathtt{x}_i \neq \mathtt{x}'_i)$ [17] and relates variables of two time frames (for example, for $k = 2$ and the path $\pi = [\mathtt{x}_1 < 0]; \mathtt{x}_1 = \mathtt{x}_2 + 1$ we obtain $(\mathtt{x}_1 < 0) \wedge (\mathtt{x}'_1 = \mathtt{x}_2 + 1) \wedge (x'_2 = \mathtt{x}_2)$).

### 2.2 A Motivating Example

A common characteristic of many contemporary symbolic software model checking techniques (such as counterexample-guided abstraction refinement with predicate abstraction [1, 8], lazy abstraction with interpolants [16], and bounded model checking [4]) is that the computational effort required to discover an assertion violation may increase exponentially with the length of the corresponding counterexample path (c.f. [13]). In particular, the detection of assertion violations that require a large number of loop iterations results in the enumeration of increasingly longer *spurious* counterexamples traversing that loop. This problem is illustrated by the following example.

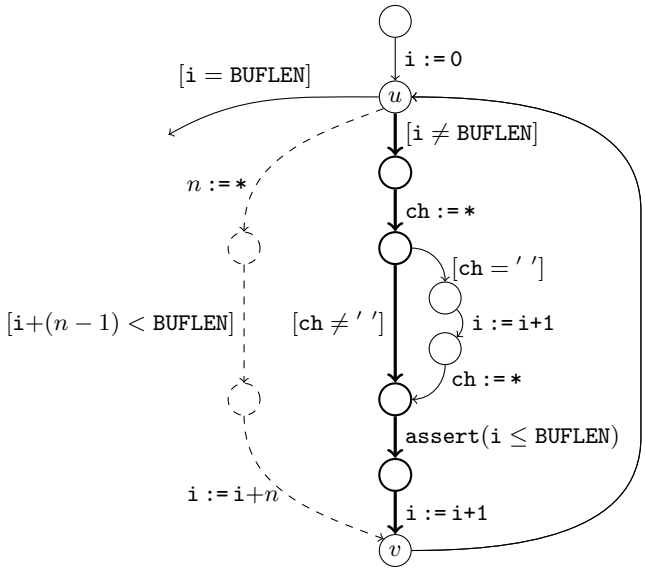



Fig. 1: CFG with path $\pi$ (bold) and approximated path $\overset{\rightsquigarrow}{\pi}$ (dashed)

*Example 1.* Figure 1 shows a program fragment derived from code permitting a buffer overflow (detected by the assertion) to occur in the $n^{\text{th}}$ iteration of the loop if i reaches (BUFLEN $-1$) and the branch [ch $=$ ' '] is taken in the $(n-1)^{\text{th}}$ iteration. The verification techniques mentioned above explore the paths in order of increasing length. The shortest path that reaches the assertion does not violate it, as

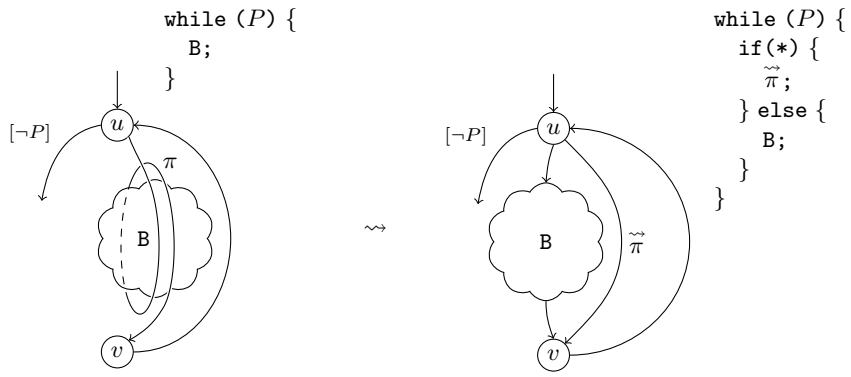
$$sp((\texttt{i} := 0; [\texttt{i} \neq \texttt{BUFLEN}]; \texttt{ch} := *; [\texttt{ch} \neq \text{' '}]), \mathsf{T}) \quad \Rightarrow \quad (\texttt{i} \leq \texttt{BUFLEN})\,.$$

In a predicate abstraction or lazy abstraction framework, this path represents the first in a series of spurious counterexamples of increasing length. Let $\pi$ denote the path emphasised in Figure 1, which traverses the loop once. The verification tool will generate a family of spurious counterexamples with the prefixes i := 0; $\pi^n$ (where $0 < n \leq \frac{\texttt{BUFLEN}}{2}$) before it detects a path long enough to violate the assertion. Each of these paths triggers a computationally expensive refinement cycle. Similarly, a bounded model checker will fail to detect a counterexample unless the loop bound is increased to $\frac{\texttt{BUFLEN}}{2} + 1$.

The iterative exploration of increasingly deeper loops primarily delays the detection of assertion violations (c.f. [13]), but can also result in a diverging series of interpolants and predicates if the program is safe (see [10]).

### 2.3 Approximating Paths with Loops

We propose a technique that aims at avoiding the enumeration of paths with an insufficient number of loop iterations. Our approach is based on the insight that




Detect path $\pi$ that repeatedly traverses the loop body `B`

Augment loop body with a branch containing the *approximation* $\overset{\rightsquigarrow}{\pi}$

Fig. 2: Approximating the natural loop with head $u$ and back-edge $v \to u$. Path $\pi$ is a path traversing the body `B` at least once, and may take different branches in `B` in subsequent iterations.

the refutation of spurious counterexamples containing a sub-path of the form $\pi^n$ is futile if there exists an $n$ large enough to permit an assertion violation. We add an auxiliary path that bypasses the original loop body and represents the effect of $\pi^n$ for a range of $n$ (detailed later in the paper). Our approach comprises the following steps:

1. We sensitise an existing tool to detect paths $\pi$ that repeatedly traverse the loop body `B` (as illustrated in the left half of Figure 2). We emphasise that $\pi$ may span more than one iteration of the loop, and that the branches of `B` taken by $\pi$ in different iterations may vary.
2. We construct a path $\overset{\rightsquigarrow}{\pi}$ whose behaviour *under-approximates* $\square\{\pi^n \mid n \geq 0\}$. This construction does not correspond to acceleration in a strict sense, since $\overset{\rightsquigarrow}{\pi}$ (as an under-approximation) does not necessarily represent an arbitrary number of loop iterations. §3 describes techniques to derive $\overset{\rightsquigarrow}{\pi}$.
3. By construction, the assumptions in $\overset{\rightsquigarrow}{\pi}$ may contain universal quantifiers ranging over an auxiliary variable which encodes the number of loop iterations. In §4, we discuss two cases in which (some of) these quantifiers can be eliminated, namely (a) if the characteristic function of the predicate $\neg wlp(\pi^n, \mathsf{F})$ is *monotonic* in the number of loop iterations $n$, or (b) if $\pi^n$ modifies an array and the indices of the modified array elements can be characterised by means of a quantifier-free predicate. We show that in certain cases condition (a) can be met by splitting $\pi$ into several separate paths.
4. We augment the control flow graph with an additional branch of the loop containing $\overset{\rightsquigarrow}{\pi}$ (Figure 2, right). §5 demonstrates empirically how this program transformation can accelerate the detection of bugs that require a large number of loop iterations.

The following example demonstrates how our technique accelerates the detection of the buffer overflow of Example 1.

*Example 2.* Assume that the verification tool encounters the node $u$ in Figure 1 a second time during the exploration of a path ($u$ is the head of a natural loop with back-edge $v \to u$). We conclude that there exists a family of (sub-)paths $\pi^n$ induced by the number $n$ of loop iterations. The repeated application of the strongest post-condition to the parametrised path $\pi^n$ for an increasing $n$ gives rise to a recurrence equation $\mathtt{i}^{\langle n \rangle} = \mathtt{i}^{\langle n-1 \rangle} + 1$ (for clarity, we work on a sliced path omitting statements referring to `ch`):

$$\begin{aligned}
sp(\pi^1, \mathsf{T}) &= \exists \mathtt{i}^{\langle 0 \rangle} \,.\, (\mathtt{i}^{\langle 0 \rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i} = \mathtt{i}^{\langle 0 \rangle} + 1) \\
sp(\pi^2, \mathsf{T}) &= \exists \mathtt{i}^{\langle 0 \rangle}, \mathtt{i}^{\langle 1 \rangle} \,.\, (\mathtt{i}^{\langle 0 \rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i}^{\langle 1 \rangle} < \mathtt{BUFLEN}) \wedge \\
&\qquad\qquad (\mathtt{i}^{\langle 1 \rangle} = \mathtt{i}^{\langle 0 \rangle} + 1) \wedge (\mathtt{i} = \mathtt{i}^{\langle 1 \rangle} + 1) \\
&\;\;\vdots \\
sp(\pi^n, \mathsf{T}) &= \exists \mathtt{i}^{\langle 0 \rangle} \ldots \mathtt{i}^{\langle n-1 \rangle} \,.\, \left( \textstyle\bigwedge_{j=0}^{n-1} (\mathtt{i}^{\langle j \rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i}^{\langle j+1 \rangle} = \mathtt{i}^{\langle j \rangle} + 1) \right)
\end{aligned}$$

where $\mathtt{i}^{\langle n \rangle}$ in the last line represents $\mathtt{i}$ after the execution of $\pi^n$. This recurrence equation can be put into its equivalent closed form $\mathtt{i}^{\langle n \rangle} = \mathtt{i}^{\langle 0 \rangle} + n$. By assigning $n$ a (positive) non-deterministic value, we obtain the approximation (which happens to be exact in this case):

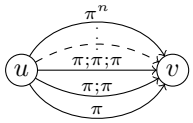
$$\overset{\rightsquigarrow}{\pi} \quad = \quad n := *; [\forall j \in [0, n) \,.\, \mathtt{i} + j < \mathtt{BUFLEN}]; \mathtt{i} := \mathtt{i} + n \quad .$$

Let us ignore arithmetic over- or under-flow for the time being (this topic is addressed in §3.4). We can then observe the following: if the predicate $\mathtt{i} + j < \mathtt{BUFLEN}$ is true for $j = n - 1$, then it must be true for any $j < n - 1$, i.e., the characteristic function of the predicate is *monotonic* in its parameter $j$. It is therefore possible to eliminate the universal quantifier and replace the assumption in $\overset{\rightsquigarrow}{\pi}$ with $(\mathtt{i} + (n - 1) < \mathtt{BUFLEN})$. The dashed path in Figure 1 illustrates the corresponding modification of the original program. The resulting transformed program permits the violation of the assertion in the original loop body after a single iteration of $\overset{\rightsquigarrow}{\pi}$ (corresponding to `BUFLEN`-1 iterations of $\pi$).

The following presents techniques to compute the under-approximation $\overset{\rightsquigarrow}{\pi}$.

## 3 Under-Approximation Techniques

This section covers techniques to compute under-approximations $\overset{\rightsquigarrow}{\pi}$ of $\bigsqcap\{\pi^n \mid n \geq 0\}$ such that $\overset{\rightsquigarrow}{\pi}$ is a condensation of the CFG fragment to the right.




The construction of $\overset{\rightsquigarrow}{\pi}$ has two aspects. Firstly, we need to make sure that all variables modified in $\overset{\rightsquigarrow}{\pi}$ are assigned values consistent with $\pi^n$ for a non-deterministic choice of $n$. Secondly, $\overset{\rightsquigarrow}{\pi}$ must only allow choices of $n$ for which $\neg wlp(\pi^n, \mathsf{F})$ is satisfiable, i.e., the corresponding path $\pi^n$ must be *feasible*.

Our approximation technique is based on the observation that the sequence of assignments in $\pi^n$ to a variable $\mathtt{x} \in \mathtt{X}$ corresponds to a recurrence equation (c.f. Example 2). The goal is to derive an equivalent *closed* form $\mathtt{x} := f_{\mathtt{x}}(\mathtt{X}, n)$. While there is a range of techniques to solve recurrence equations, we argue that it is sufficient to consider closed-form solutions that have the form of low-degree polynomials. The underlying argument is that a super-polynomial growth of variable values typically leads to an arithmetic overflow after a small number of iterations, which can be detected at low depth using conventional techniques.

The following sub-section focuses on deriving closed forms from a sequence of assignments to scalar integer variables, leaving conditionals aside. §3.2 covers assignments to arrays. Conditionals and path feasibility are addressed in §3.3. §3.4 addresses bit-vector semantics and arithmetic overflow.

### 3.1 Computing Closed Forms of Assignments

**Syntactic Matching.** A simple technique to derive closed forms is to check whether the given recurrence equation matches a pre-determined format. In our work on loop detection for predicate abstraction [13, 14], we apply the following scheme:

$$\mathtt{x}^{\langle 0\rangle} = \alpha, \quad \mathtt{x}^{\langle n\rangle} = \mathtt{x}^{\langle n-1\rangle} + \beta + \gamma \cdot n \quad \rightsquigarrow \quad \mathtt{x}^{\langle n\rangle} = \alpha + \beta n + \gamma \frac{n \cdot (n+1)}{2}\,, \tag{1}$$

where $n > 0$ and $\alpha$, $\beta$, and $\gamma$ are numeric constants or loop-invariant *symbolic* expressions and $\mathtt{x}$ is the variant. This technique is computationally cheap and sufficient to construct the closed form $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle 0\rangle} + n$ of the recurrence equation $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle n-1\rangle} + 1$ derived from the assignment $\mathtt{i} := \mathtt{i} + 1$ in Example 2.

**Constraint-based Acceleration.** The disadvantage of a syntax-based approach is that it is limited to assignments following a simple pattern. Moreover, the technique is contingent on the syntax of the program fragment and may therefore fail even if there *exists* an appropriate polynomial representing the given assignments. In this section, we present an alternative technique that relies on a constraint solver to identify the coefficients of the polynomial $f_{\mathtt{x}}$.

Let $\mathtt{X}$ be the set $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ of variables in $\pi$. (In the following, we omit the braces $\{\}$ if clear from the context.) As previously, we start with the assumption that for each variable $\mathtt{x}$ modified in $\pi$, there is a low-degree polynomial in $n$

$$f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, n) \stackrel{\text{def}}{=} \sum_{i=1}^{k} \alpha_i \cdot \mathtt{x}_i^{\langle 0\rangle} + \left( \sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)} \right) \cdot n + \alpha_{(2\cdot k+2)} \cdot n^2 \tag{2}$$

over the initial variables $\mathtt{x}_1^{\langle 0\rangle}, \ldots, \mathtt{x}_k^{\langle 0\rangle}$ which accurately represents the value assigned to $\mathtt{x}$ in $\pi^n$ (for $n \geq 1$). In other words, for each variable $\mathtt{x} \in \mathtt{X}$ modified in $\pi$, we assume that the following Hoare triple is valid:

$$\{\bigwedge_{i=1}^{k} \text{`}\mathtt{x}_i = \mathtt{x}_i\}\ \pi^n\ \{\mathtt{x} = f_{\mathtt{x}}(\text{`}\mathtt{x}_1, \ldots, \text{`}\mathtt{x}_k, n)\} \tag{3}$$

For each $\mathtt{x} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ we can generate $2 \cdot k + 2$ distinct assignments to $\mathtt{x}_1^{\langle 0 \rangle}, \ldots, \mathtt{x}_k^{\langle 0 \rangle}$, and $n$ in (2) which determine a system of linearly independent equations over $\alpha_i$, $0 < i \leq 2 \cdot k + 2$ (this can be proven by induction on $k$). If a solution to this system of equations exists, it *uniquely* determines the parameters $\alpha_1, \ldots, \alpha_{2 \cdot k+2}$ of the polynomial $f_{\mathtt{x}}$ for $\mathtt{x}$. In particular, the satisfiability of the encoding from which we derive the assignments guarantees that (3) holds for $0 \leq n \leq 2$. For larger values of $n$, we check the validity of (3) with respect to each $f_{\mathtt{x}}$ by means of induction. The validity of (3) follows (by induction over the length of the path $\pi^n$) from the validity of the base case established above, the formula (4) given below (which can be easily checked using a model checker or a constraint solver), and Hoare's rule of composition:

$$\{\bigwedge_{i=1}^{k} \left(\grave{}\mathtt{x}_i = \mathtt{x}_i\right) \wedge \mathtt{x} = f_{\mathtt{x}}(\grave{}\mathtt{x}_1, \ldots, \grave{}\mathtt{x}_k, n)\} \; \pi \; \{\mathtt{x} = f_{\mathtt{x}}(\grave{}\mathtt{x}_1, \ldots, \grave{}\mathtt{x}_k, n+1)\} \tag{4}$$

If for one or more $\mathtt{x} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ our technique fails to find valid parameters $\alpha_1, \ldots, \alpha_{2 \cdot k+2}$, or the validity check for $f_{\mathtt{x}}$ fails, we do not construct $\overset{\rightsquigarrow}{\pi}$.

*Remark.* The construction of under-approximations is not limited to the two techniques discussed above and can be based on other (and potentially more powerful) recurrence solvers.

### 3.2 Assignments to Arrays

Buffer overflows constitute a prominent class of safety violations that require a large number of loop iterations to surface. In C programs, buffers and strings are typically implemented using arrays. Let $\mathtt{i}$ be the variant of a loop which contains an assignment $\mathtt{a[i]:=}e$ to an array $\mathtt{a}$. For a single iteration, we obtain

$$sp(\mathtt{a[i]} := e, P) \stackrel{\text{def}}{=} \exists \grave{}\mathtt{a} \,.\, \mathtt{a[i]} = e[\mathtt{a}/\grave{}\mathtt{a}] \wedge \forall j \neq \mathtt{i} \,.\, \left(\mathtt{a}[j] = \grave{}\mathtt{a}[j]\right) \wedge P[\mathtt{a}/\grave{}\mathtt{a}] \tag{5}$$

Assume further that closed forms for the variant $\mathtt{i}$ and the expression $e$ exist (abusing our notation, we use $f_e$ to refer to the latter). Given an initial pre-condition $P = \mathsf{T}$, we obtain the following *approximation*[3] after $n$ iterations:

$$\begin{aligned} &\forall j \in [0, n) \,.\, \mathtt{a}^{\langle n \rangle}[f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j)] = f_e(\mathtt{X}^{\langle 0 \rangle}, j) \wedge \\ &\qquad \forall i \in \operatorname{dom} \mathtt{a} \,.\, \underbrace{\left(\exists j \in [0, n) \,.\, i = f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j)\right)}_{\text{membership test}} \vee \left(\mathtt{a}^{\langle n \rangle}[i] = \mathtt{a}^{\langle 0 \rangle}[i]\right) , \end{aligned} \tag{6}$$

where the domain ($\operatorname{dom} \mathtt{a}$) of $\mathtt{a}$ denotes the valid indices of the array. Notably, the membership test determining whether an array element is modified or not introduces quantifier alternation, posing a challenge to contemporary decision procedures. §4 addresses the elimination of the existential quantifier in (6).

[3] Condition (6) *under*-approximates the strongest post-condition, since there may exist $j_1, j_2 \in [0, n)$ such that $j_1 \neq j_2 \wedge f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j_1) = f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j_2)$ and (6) is unsatisfiable. A similar situation arises if a loop body $\pi$ contains multiple updates of the same array.

### 3.3 Assumptions and Feasibility of Paths

The techniques discussed in §3.1 yield polynomials and constraints representing the assignment statements of $\pi^n$, but leave aside the conditional statements which determine the feasibility of the path. In the following, we demonstrate how to derive the pre-condition $\neg wlp(\pi^n, \mathsf{F})$ using the polynomials $f_{\mathtt{x}}$ for $\mathtt{x} \in \mathtt{X}$.

Let $f_{\mathtt{X}}(\mathtt{X}, n) \stackrel{\text{def}}{=} \{f_{\mathtt{x}}(\mathtt{X}, n) \,|\, \mathtt{x} \in \mathtt{X}\}$ and let $Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, n)]$ denote the simultaneous substitution of all free occurrences of the variables $\mathtt{x} \in \mathtt{X}$ in $Q$ with the corresponding term $f_{\mathtt{x}}(\mathtt{X}, n)$.

**Lemma 1.** *The following equivalence holds:*

$$wlp(\pi^n, \mathsf{F}) \quad \equiv \quad \exists j \in [0, n) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]$$

*Proof.* Intuitively, the path $\pi^n$ is infeasible if for *any* $j < n$ the first time-frame of the suffix $\pi^{(n-j)}$ is infeasible. We prove the claim by induction over $n$. Due to (3) and (4) we have $f_{\mathtt{X}}(\mathtt{X}, 0) = \mathtt{X}$ and $f_{\mathtt{X}}(f_{\mathtt{X}}(\mathtt{X}, n), 1) = f_{\mathtt{X}}(\mathtt{X}, n+1)$ (for $n \geq 0$).

*Base case:* $wlp(\pi, \mathsf{F}) \equiv \exists j \in [0, 0) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)] = \mathsf{F}$

*Induction step.* We start by applying the induction hypothesis:

$$\begin{aligned} wlp(\pi^n, \mathsf{F}) &\equiv wlp(\pi, \big(wlp(\pi^{n-1}, \mathsf{F})\big)) \\ &\equiv wlp(\pi, \exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]) \end{aligned}$$

We consider the effect of assignments and assumptions occurring in $\pi$ on the post-condition $Q \stackrel{\text{def}}{=} (\exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)])$ separately.

– The effect of assignments in $\pi$ on $Q$ is characterised by $Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, 1)]$. We obtain:

$$\begin{aligned} Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, 1)] \;\equiv\; \exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(f_{\mathtt{X}}(\mathtt{X}, 1), j)] \;\equiv \\ \exists j \in [1, n) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)] \end{aligned}$$

– Assumptions in $\pi$ contribute the disjunct $wlp(\pi, \mathsf{F})$.

By combining both contributions into one term we obtain

$$wlp(\pi^n, \mathsf{F}) \equiv (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, 0)] \vee \exists j \in [1, n) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]\,,$$

which establishes the claim of Lemma 1.

Accordingly, given a path $\pi$ modifying the set of variables $\mathtt{X}$ and a corresponding set $f_{\mathtt{X}}$ of closed-form assignments, we can construct an accurate representation of $\pi^n$ as follows:

$$\underbrace{[\,\forall j \in [0, n) \,.\, (\neg wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]\,]}_{\text{satisfiable if } \pi^n \text{ is feasible}} \quad ; \quad \underbrace{\mathtt{X} := f_{\mathtt{X}}(\mathtt{X}, n)}_{\text{assignments of } \pi^n} \tag{7}$$

We emphasise that our construction (unlike many acceleration techniques) does *not* restrict the assumptions in $\pi$ to a limited class of relations on integers.

The construction of the path (7), however, does require closed forms of all assignments in $\pi$. Since we do not construct closed forms for array assignments (as opposed to assignments to array indices, c.f. §3.2), we cannot apply Lemma 1 if $wlp(\pi, \mathsf{F})$ refers to an array assigned in $\pi$. In this case, we do not construct $\overset{\rightsquigarrow}{\pi}$.

For assignments of variables not occurring in $wlp(\pi, \mathsf{F})$, we augment the domain(s) of the variables $\mathtt{X}$ with an undefined value $\perp$ (implemented using a Boolean flag) and replace $f_{\mathtt{x}}$ with $\perp$ whenever the respective closed form is not available. Subsequently, whenever the search algorithm encounters an (abstract) counterexample, we use slicing to determine whether the feasibility of the counterexample depends on an undefined value $\perp$. If this is the case, the counterexample needs to be dismissed. Thus, any path $\overset{\rightsquigarrow}{\pi}$ containing references to $\perp$ is an *under-approximation* of $\pi^n$ rather than an acceleration of $\pi$.

*Example 3.* For a path $\pi \overset{\text{def}}{=} [\mathtt{x} < 10]; \mathtt{x} := \mathtt{x} + \mathtt{1}; \mathtt{y} := \mathtt{y}^2$, we obtain the under-approximation $\overset{\rightsquigarrow}{\pi} \equiv n := *; [\forall j \in [0, n).\mathtt{x} + j < 10]; \mathtt{x} := \mathtt{x} + n; \mathtt{y} := \perp$. A counterexample traversing $\overset{\rightsquigarrow}{\pi}$ is feasible if its conditions do not depend on $\mathtt{y}$.

### 3.4 Arithmetic Overflows

The fact that the techniques in §3.1 used to derive closed forms do not take arithmetic overflow into account may lead to undesired effects. For instance, the assumption made in Example 2 that the characteristic function of the predicate $(\mathtt{i} + n < \mathtt{BUFLEN})$ is monotonic in $n$ does not hold in the context of bit-vectors or modular arithmetic. Since, moreover, the behaviour of arithmetic over- or under-flow in C is not specified in certain cases, we conservatively rule out all occurrences thereof in $\overset{\rightsquigarrow}{\pi}$. For the simple assignment $\mathtt{i} := \mathtt{i} + n$ in Example 2, this can be achieved by adding the assumption $(\mathtt{i} + n \leq 2^l - 1)$ to $\overset{\rightsquigarrow}{\pi}$ (for unsigned $l$-bit vectors). In general, we have to add respective assumptions $(e_1 \otimes e_2 \leq 2^l - 1)$ for all arithmetic (sub-)expressions $e_1 \otimes e_2$ of bit-width $l$ and operations $\otimes$ in $\overset{\rightsquigarrow}{\pi}$.

While this approach is *sound* (eliminating paths from $\overset{\rightsquigarrow}{\pi}$ does not affect the correctness of the instrumented program, since all behaviours following an overflow are still reachable via non-approximated paths), it imposes restrictions on the range of $n$. Therefore, the resulting approximation $\overset{\rightsquigarrow}{\pi}$ deviates from the acceleration $\pi^*$ of $\pi$. Unlike acceleration over linear affine relations, this adjustment makes our approach bit-level accurate. We emphasise that the benefit of the instrumentation can still be substantial, since the number of iterations required to trigger an arithmetic overflow is typically large.

## 4 Eliminating Quantifiers from Approximations

A side effect of the approximation steps in §3.2 and §3.3 is the introduction of quantified assumptions. While quantification is often unavoidable in the presence of arrays, it is a detriment to performance of the decision procedures underlying

the verification tools. In the worst case, quantifiers may result in the undecidability of path feasibility.

In the following, we discuss two techniques to eliminate or reduce the number of quantifiers in assumptions occurring in $\overset{\rightsquigarrow}{\pi}$.

### 4.1 Eliminating Quantifiers over Monotonic Predicates

We show that the quantifiers introduced by the technique presented in §3.3 can be eliminated if the predicate is monotonic in the quantified parameter.

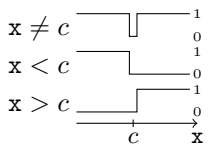
**Definition 1 (Representing Function, Monotonicity).** *The representing function $f_P$ of a predicate $P$ with the same domain takes, for each domain value, the value 0 if the predicate holds, and 1 if the predicate evaluates to false, i.e., $P(\mathtt{X}) \Leftrightarrow f_P(\mathtt{x}) = 0$. A predicate $P(n) : \mathbb{N} \to \mathbb{B}$ is monotonically increasing (decreasing) if its representing function $f_P(n) : \mathbb{N} \to \mathbb{N}$ is monotonically increasing (decreasing), i.e., $\forall m, n \,.\, m \leq n \Rightarrow f_P(m) \leq f_P(n)$.*

We extend this definition to predicates over variables $\mathtt{X}$ and $n \in \mathbb{N}$ as follows: $P(\mathtt{X}, n)$ is monotonically increasing in $n$ if $(m \leq n) \wedge P(\mathtt{X}, n) \wedge \neg P(\mathtt{X}, m)$ is unsatisfiable.

**Proposition 1.** *$P(\mathtt{X}, n-1) \equiv \forall i \in [0, n) \,.\, P(\mathtt{X}, i)$ if $P$ is monotonically increasing in $i$.*

The validity of Proposition 1 follows immediately from the definition of monotonicity. Accordingly, it is legitimate to replace universally quantified predicates in $\overset{\rightsquigarrow}{\pi}$ with their corresponding unquantified counterparts (c.f. Proposition 1).

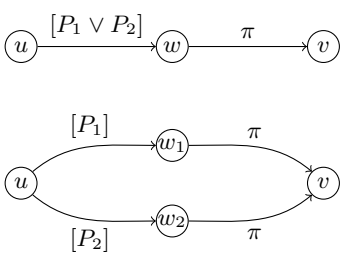
This technique, however, fails for simple cases such as $\mathtt{x} \neq c$ ($c$ being a constant). In certain cases, the approach can still be applied after *splitting* a non-monotonic predicate $P$ into monotonic predicates $\{P_1, \ldots, P_m\}$ such that $P \equiv \bigvee_{i=1}^{m} P_i$ (as illustrated in the Figure to the right). Subsequently, the path $\pi$ guarded by $P$ can be split as outlined in Figure 3. This transformation preserves reachability (a proof for $m = 2$ is given in Figure 3).




This approach is akin to trace partitioning [7], however, our intent is quantifier elimination rather than refining an abstract domain. We rely on a template-based approach to identify predicates that can be split (a constraint solver-based approach is bound to fail if $c$ is symbolic). While this technique effectively deals with a broad number of standard cases, it does fail for quantifiers over array indices, since the array access operation is not monotonic.

### 4.2 Eliminating Quantifiers in Membership Tests for Array Indices

This sub-section aims at replacing the existentially quantified membership test in Predicate (6) by a quantifier-free predicate. To define a set of sufficient (but not necessary) conditions for when this is possible, we introduce the notion of increasing and dense array indices (c.f. [11]):




$$sp([P_1 \vee P_2]; \pi, Q) \equiv sp(\pi, (P_1 \wedge Q) \vee (P_2 \wedge Q)) \equiv sp(\pi, P_1 \wedge Q) \vee sp(\pi, P_2 \wedge Q)$$

$$sp(([P_1]; \pi) \sqcap ([P_2]; \pi), Q) \equiv sp([P_1]; \pi, Q) \vee sp([P_2]; \pi, Q) \equiv sp(\pi, (P_1 \wedge Q)) \vee sp(\pi, (P_2 \wedge Q))$$

Fig. 3: Splitting disjunctive assumptions preserves program behaviour

**Definition 2 (Increasing and Dense Variables).** *A scalar variable* $\mathtt{x}$ *is (strictly)* increasing *in* $\pi^n$ *iff* $\forall j \in [0, n) \,.\, \mathtt{x}^{\langle j+1 \rangle} \geq \mathtt{x}^{\langle j \rangle}$ *(*$\forall j \in [0, n) \,.\, \mathtt{x}^{\langle j+1 \rangle} > \mathtt{x}^{\langle j \rangle}$*, respectively). Moreover, an increasing variable* $\mathtt{i}$ *is* dense *iff*

$$\forall j \in [0, n) \,.\, \left(\mathtt{x}^{\langle j+1 \rangle} = \mathtt{x}^{\langle j \rangle}\right) \vee \left(\mathtt{x}^{\langle j+1 \rangle} = \mathtt{x}^{\langle j \rangle} + 1\right) .$$

*Variables decreasing in* $\pi^n$ *are defined analogously. A variable is monotonic (in* $\pi^n$*) if it is increasing or decreasing (in* $\pi^n$*).*

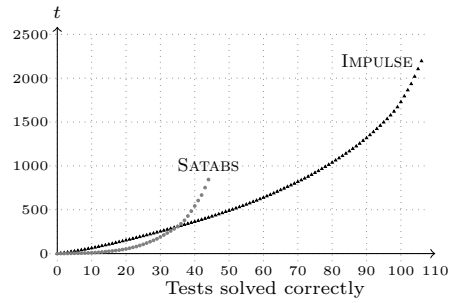
Note that if the closed form $f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, n)$ of a variable $\mathtt{x}$ is a linear polynomial, then $\mathtt{x}$ is necessarily monotonic. The following proposition uses this property:

**Proposition 2.** *Let* $f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j)$ *be the closed form (2) of* $\mathtt{x}^{\langle j \rangle}$*, where* $\alpha_{(2 \cdot k+2)} = 0$*, i.e., the polynomial* $f_{\mathtt{x}}$ *is linear. Then* $\Delta f_{\mathtt{x}} \stackrel{\text{def}}{=} f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j+1) - f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j)$ *(for* $j \in [0, n)$*) is the (symbolic) constant* $\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i^{\langle 0 \rangle} + \alpha_{(2 \cdot k+1)}$*. The variable* $\mathtt{x}$ *is (strictly) increasing in* $\pi^n$ *if* $\Delta f_{\mathtt{x}} \geq 0$ *(*$\Delta f_{\mathtt{x}} > 0$*, respectively) and dense if* $0 \leq \Delta f_{\mathtt{x}} \leq 1$*.*

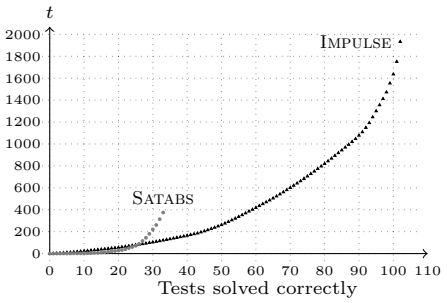
**Lemma 2.** *Let* $f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j)$ *be a linear polynomial representing the closed form (2) of* $\mathtt{x}^{\langle j \rangle}$ *(as in Proposition 2). The following logical equivalence holds:*

$$\exists j \in [0, n) \,.\, \mathtt{x} = f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j) \;\equiv\; \begin{cases} ((\mathtt{x} - \mathtt{x}^{\langle 0 \rangle}) \mod \Delta f_{\mathtt{x}} = 0) \wedge (\frac{\mathtt{x} - \mathtt{x}^{\langle 0 \rangle}}{\Delta f_{\mathtt{x}}} < n) & \text{if } \mathtt{x} \text{ is strictly increasing} \\ \mathtt{x} - \mathtt{x}^{\langle 0 \rangle} \leq (n-1) \cdot \Delta f_{\mathtt{x}} & \text{if } \mathtt{x} \text{ is dense} \\ \mathtt{x} - \mathtt{x}^{\langle 0 \rangle} < n & \text{if both of the above hold} \end{cases} \tag{8}$$

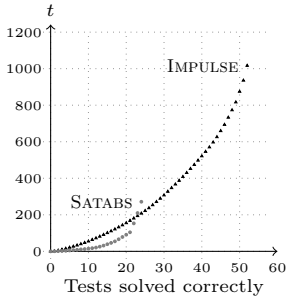
The validity of Lemma 2 follows immediately from Proposition 2. Using Lemma 2, we can replace the existentially quantified membership test in Predicate (6) by a quantifier-free predicate if one of the side conditions in (8) holds. Given that the path prefix reaches the entry node of a loop, these conditions $\Delta f_{\mathtt{x}} > 0$ and $0 \leq \Delta f_{\mathtt{x}} \leq 1$ can be checked using a satisfiability solver.



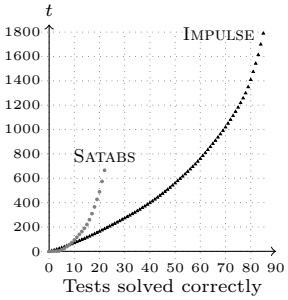

(a) Safe and unsafe, buffer size 10



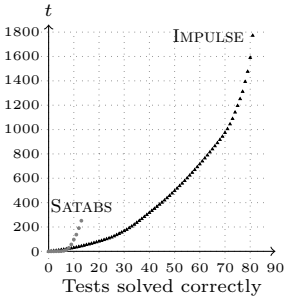

(b) Safe and unsafe, buffer size $10^2$




(c) Safe/unsafe, b.-size $10^3$




(d) Unsafe, buffer size 10




(e) Unsafe, buffer size $10^2$

Fig. 4: Verification run-times (cumulative) of VERISEC benchmark suite

*Example 4.* Let $\pi \stackrel{\text{def}}{=} \mathtt{a}[\mathtt{x}] := \mathtt{x};\ \mathtt{x} := \mathtt{x} + 1$ be the body of a loop. By instantiating (6), we obtain the condition

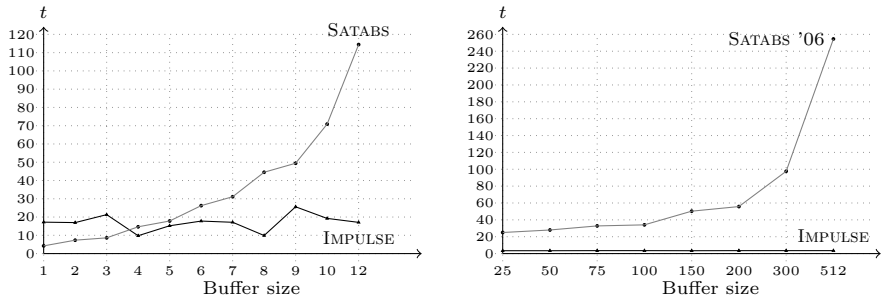
$$\forall j \in [0, n)\,.\mathtt{a}[\mathtt{`x} + j] = \mathtt{`x} + j \ \wedge \forall i.\ \big(\exists j \in [0, n)\,.\, i = \mathtt{`x} + j\big) \ \vee \ \big(\mathtt{a}[i] = \mathtt{`a}[i]\big)\,,$$

in which the existentially quantified term can be replaced by $\mathtt{x} - \mathtt{`x} < n$.

## 5 Implementation and Experimental Results

Our under-approximation technique is designed to extend existing verifiers. To demonstrate its versatility, we implemented IMPULSE, a tool combining under-approximation with the two popular software verification techniques lazy abstraction with interpolants (LAWI) [16] and bounded model checking (specifically, CBMC [4]). The underlying SMT solver used throughout was version 4.2 of Z3. IMPULSE comprises two phases:

1. IMPULSE first explores the paths of the CFG following the LAWI paradigm. If IMPULSE encounters a path containing a loop with body $\pi$, it computes $\overset{\rightsquigarrow}{\pi}$




(a) Single VERISEC test, varying buffer size (b) SATABS w. loop detect on Aeon 0.02a

Fig. 5: Run-time dependency on buffer size for unsafe benchmarks

(processing inner loops first in the presence of nested loops), augments the CFG accordingly, and proceeds to phase 2.

2. CBMC inspects the instrumented CFG up to an iteration bound of 2. If no counterexample is found, IMPULSE returns to phase 1.

In phase 1, spurious counterexamples serve as a catalyst to refine the current approximation of safely reachable states, relying on the weakest precondition[4] to generate the required Hoare triples. Phase 2 takes advantage of the aggressive path merging performed by CBMC, enabling fast counterexample detection.

We evaluated the effectiveness of under-approximation on the VERISEC benchmark suite [15], which consists of manually sliced versions of several open source programs that contain buffer overflow vulnerabilities. Of the 284 test cases of VERISEC, 144 are labelled as containing a buffer overflow, and 140 are labelled as safe.[5] The safety violations range from simple unchecked string copy into static buffers, up to complex loops with pointer arithmetic. The buffer size in each benchmark (c.f. `BUFLEN` in Figure 1) is adjustable and controls the depth of the counterexample. We compared our tool with SATABS (which outperforms IMPULSE w/o approximation[6]) on buffer sizes of 10, 100 and 1000, with a time limit of 300 s and a memory limit of 2 GB on an 8-core 3 GHz Xeon CPU. Figures 4a through 4c show the cumulative run-time for the whole benchmark suite, whereas Figures 4d and 4e show only unsafe program instances. Under-approximation did not improve (or impair) the run-time on safe instances.[6]

Figure 5 demonstrates that the time IMPULSE requires to detect a buffer overflow does not depend on the buffer size. Figure 5a compares SATABS and IMPULSE on a single VERISEC benchmark with a varying size parameter, showing that SATABS takes time exponential in the size of the buffer. Figure 5b provides

[4] In a preliminary interpolation-based implementation, Z3 was in many cases unable to provide interpolants for path formulas $\overset{\rightsquigarrow}{\pi}$ with quantifiers, arrays, and bit-vectors.

[5] Our new technique discovered bugs in 10 of the benchmarks that had been labelled safe. SATABS timed out before identifying these bugs.

[6] The respective results are available on http://www.cprover.org/impulse.

a qualitative comparison of the loop-detection technique presented in [13] with IMPULSE on the Aeon 0.02a mail transfer agent. Figure 5b shows the run-times of SATABS'06 with loop detection as reported in [13],[7] as well as the run-times of IMPULSE on the same problem instances and buffer sizes. SATABS'06 outperforms similar model checking tools that do not feature loop-handling mechanisms [13]. However, the run-time still increases exponentially with the size of the buffer, since the technique necessitates a validation of the unwound counterexample. IMPULSE does not require such a validation step.

## 6 Related Work

The under-approximation technique presented in this paper is based on our previous work on loop detection [13, 14]. The algorithm in [13], however, does not yield a strict under-approximation, and thus necessitates an additional step to validate the unwound counterexample. Our new technique avoids this problem.

The techniques in §3.1 constitute a simple form of acceleration [3, 6]. The subsequent restrictions in §3.3 and §3.4 on $\overset{\rightsquigarrow}{\pi}$, however, impose a symbolic bound on the number of iterations, yielding an under-approximation. In contrast to acceleration of integer relations, our approximation is sound for bit-vector arithmetic. Sinha uses term rewriting to compute symbolic states parametrised by the loop counter, stating that his technique can be extended to support bit-vectors [19].

The quantifier elimination technique of §4.1 bears similarities with splitter predicates [18], a program transformation facilitating the generation of disjunctive invariants. Similarly, trace partitioning [7] splits program paths to increase the precision of static analyses. Neither technique aims at eliminating quantifiers.

Loop summarisation [12] and path invariants [2] avoid loop unrolling by selecting an appropriate over-approximation of the loop from a catalogue of invariant templates. Over-approximations are also used in the context of loop bound inference [20] and reasoning about termination [5]. However, over-approximations do not enable the efficient detection of counterexamples.

Hojjat et al. [9] uses interpolation to derive inductive invariants from accelerated paths. While this work combines under- and over-approximation, it is not aimed at counterexample detection. Motivated by the results of [9], we believe that under-approximation can, if combined with interpolation, improve the performance of verification tools on safe programs. We plan to support interpolation in a future version of our implementation.

We refer the reader to [14] for a description of additional related work.

## 7 Conclusion and Future Work

We present a sound under-approximation technique for loops in C programs with bit-vector semantics. The approach is very effective for finding deep counterexamples in programs that manipulate arrays, and compatible with a variety

[7] Unfortunately, loop detection in SATABS is neither available nor maintained anymore.

of existing verification techniques. A short-coming of our under-approximation technique is its lack of support for dynamic data structures, which we see as a challenging future direction.

## References


1. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *IFM*, volume 2999 of *LNCS*. Springer, 2004.
2. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309. ACM, 2007.
3. B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1999.
4. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176. Springer, 2004.
5. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, 2011.
6. A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST-TCS 2002*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.
7. M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *SAS*, volume 1503 of *LNCS*, pages 200–214. Springer, 1998.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
9. H. Hojjat, R. Iosif, F. Konecny, V. Kuncak, and P. Ruemmer. Accelerating interpolants. In *ATVA*, volume 7561 of *LNCS*, pages 197–202, 2012.
10. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.
11. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.
12. D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger. Loop summarization using abstract transformers. In *ATVA*, volume 5311 of *LNCS*, pages 111–125. Springer, 2008.
13. D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *CAV*, volume 4144 of *LNCS*, pages 152–165. Springer, 2006.
14. D. Kroening and G. Weissenbacher. Verification and falsification of programs with loops using predicate abstraction. *Formal Aspects of Computing*, 22:105–128, 2010.
15. K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392. ACM, 2007.
16. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.
17. G. Nelson. A generalization of Dijkstra's calculus. *TOPLAS*, 11(4):517–561, 1989.
18. R. Sharma, I. Dillig, T. Dillig, and A. Aiken. Simplifying loop invariant generation using splitter predicates. In *CAV*, volume 6806 of *LNCS*, pages 703–719. Springer, 2011.
19. N. Sinha. Symbolic program analysis using term rewriting and generalization. In *Formal Methods in Computer-Aided Design*, pages 1–9, 2008.
20. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.