# Stabbing Loops in the Back or How to Unroll a Loop on an Arbitrary Iteration

Mikhail Belyaev, Marat Akhin, and Vladimir Itsykson

Saint-Petersburg State Polytechnical University, Russia
{belyaev,akhin}@kspt.icc.spbstu.ru vlad@icc.spbstu.ru

**Abstract.** Program loops have proven to be a matter of great difficulty for any method of static analysis or verification, because of the state space explosion and of the well-known halting problem that prevents us from reasoning about loop iteration counts in many practical cases. The current state-of-the-art approach most widely used in code analysis tools is a simple approach we call "unroll-and-bound", i.e. unrolling the loop for a given number of iterations which can be either fixed or calculated for the given loop using trip count analyses.
In this paper we present an original approach to loop analysis for tools based on logic reasoning, e.g. bounded model checkers. It is based on unrolling loops not only from the beginning, but from the end or from any other loop iteration. Loop body is instantiated using existing compiler scalar evolution analysis, and we show that a large number of loops can be analyzed like this including, surprisingly enough, some types of infinite loops. The approach also provides some advantages when analyzing loops that can be fully unrolled using a fixed bound by greatly reducing the unrolled code size with no penalty to analysis quality.
Our approach has been implemented in the prototype bounded model checking tool called Borealis. Evaluation results on a number of test programs from NECLA and SV-COMP benchmarks show an over nine-fold increase in performance as well as some increase in analysis quality when using the presented approach.

**Keywords:** Static Program Analysis, Loop Unrolling, LLVM, Bounded Model Checking, Satisfiability Modulo Theories,

## 1 Introduction

Program analysis becomes increasingly important nowadays as one of the main ways of software quality assurance. Unfortunately, real-world software presents a lot of problems for different kinds of program analyses. One of the most important problems is how you analyze loops in a given program.

Every loops represents a potentially exponential number of possible execution paths, thus causing an exponential increase in analysis complexity. Moreover, some loops cannot be bound statically and represent a possibly infinite number of paths. Therefore, to efficiently explore all possible loop paths one has to either limit or summarize the execution state space.

In this paper we present an original approach to loop analysis that reinforces traditional loop unrolling with what we call "loop backstabbing". It analyzes a given loop not only on the first $N$ iterations, but also on some arbitrary iterations $I_1, I_2, \ldots, I_k$. To do this, we take advantage of the recent advances in compilers and leverage existing induction variable and scalar evolution analyses. Using them, one can instantiate a loop body on any given iteration, thus summarizing possible loop executions. For example, if you instantiate the loop body on the last iteration, you can check whether it contains "off-by-one" errors without the need to do full loop unrolling.

Of course, this approach is only as good as the capabilities of the underlying scalar evolution analysis. If it fails to analyze a loop, loop backstabbing cannot be applied and we have to fall-back to traditional loop unrolling. We implemented loop backstabbing in our prototype bounded model checking tool called Borealis and tested it on a number of examples from NECLA and SV-COMP benchmarks to measure the efficiency and the applicability of our approach. The results show that our approach demonstrates an over 9x increase in performance without any loss in analysis quality. About 60% of the loops from our test suite can be successfully analyzed by loop backstabbing.

The rest of the paper is organized as follows. We briefly overview the backgrounds used in section 2. The motivation for our research and our problem statement are given in section 3. Sections 4 and 5 describe loop backstabbing itself and its evaluation respectively. An overview of the related work on loop analysis is done in section 6.

## 2 Background

Loop analysis has long been an important field of study in optimizing compilers. Although it is not possible to reason about loop traits in general (due to the well known halting problem), most loops in actual programs tend to have a more or less defined structure and can be analyzed using some heuristics. Preliminary code transformations like loop unswitching, loop invariant code motion and loop reduction [1] can greatly increase the amount of information one can get from the code by examining loop structure.

Modern compilers like LLVM [15] and GCC [17] use SSA-based program models that open up new opportunities for loop analysis [18] such as scalar evolutions. A scalar evolution of a value inside the loop body holds the information on how this particular value evolves between loop iterations. Scalar evolution in compilers is primarily used to efficiently calculate loop trip counts and extract auxiliary induction variables, but it can also provide functionality such as:

– Calculating symbolic loop trip counts in the form of polynomials over loop invariants
– Calculating per-exit loop trip counts for multiple-exit loops
– Evolving any expressions over polynomially changing induction variables and loop invariants for any given iteration including (but not limited to) any polynomial expressions over loop invariants

We employ the full might of these techniques in our prototype system using a fully working scalar evolution analysis implementation from the LLVM compiler infrastructure.

## 3 Problem Statement and Motivation

### 3.1 Dealing with Loops in Program Analysis

Our research has been somewhat inspired by the survey study by Xiao et. al. [21] that focuses on main problems that arise when analyzing different kinds of loops. It mainly considers symbolic execution and its application to test generation, but the discovered highlights also hold for other research areas such as static analysis and bounded model checking.

There is a number of ways loops can be dealt with when analyzing program code. To the best of our knowledge, the most wide-spread techniques are:

- Bounded loop unrolling (also called "loop bounding" and "loop bounded iteration") which we refer to as "unroll-and-bound"
- Iterative bounded loop unrolling
- Inference of different loop invariants
- Loop abstraction and summarization

The first technique is amazingly simple and efficient compared to the other ones, but is also unsound as it involves non-semantic-preserving code transformations. Iterative bounded loop unrolling [7], just as the name implies, applies bounded loop unrolling iteratively, increasing the bound on each subsequent iteration until a given safety property is proven to always hold (in some cases this process never terminates). Inference of loop invariants, while very powerful, is undecidable in general and often cannot be used automatically for a large number of loop types [8]. The last technique [14] is based on finding a summary of the loop body using interpolation or other means and, like loop invariant inference, is somewhat limited in practice due to the limitations of current summarization tools.

It is worth noting that most of the existing software analysis tools (e.g., Saturn [2], LLBMC [16] or CBMC [9]) simply employ variations of the "unroll-and-bound" technique, because it is applicable to any kind of loops and is very efficient performance-wise.

That is why we decided to try and devise an approach that goes somewhat beyond simple loop unrolling, but does not sacrifice simplicity, generality and performance. To do that, we focused our attention on facilities already existing in industrial compilers, not unlike our previous work on bounded model checking [3].

### 3.2 A Closer Look on the "Unroll-and-bound" Technique

Loop unrolling is a very common code optimization technique based on rewriting a loop to a sequence of loop body statements repeated $N$ times where $N$ is a number of unrolled loop iterations. This approach is also widely used in different code analyses, as it eliminates the loop and makes the subsequent analysis much easier.

If loop iteration count is constantly known at analysis time, full loop unrolling is semantic-preserving and can be safely used to avoid the need to do loop analysis. In case loop iteration count is not constantly known, one can keep an arbitrary number of iterations and use it as a bound for unrolling, hence the name "unroll-and-bound".

Let us demonstrate this approach on two simple examples[1]. Example A contains a loop with a simple statically-known loop bound, while a loop from Example B is somewhat more complex.

**Listing 3.1.** Example A

```
int* a = malloc(sizeof(int)*3);
for(int i = 0; i <= 3; ++i) {
    a[i] = 0; // Oops, buffer overflow on the last iteration
}
```

It unrolls to:

**Listing 3.2.** Example A, fully unrolled

```
int* a = malloc(sizeof(int)*3);
// note that there are 4 iterations, and not 3
int i0 = 0;
if(i0 <= 3) {
    a[i0] = 0;
    int i1 = i0 + 1;
    if(i1 <= 3) {
        a[i1] = 0;
        int i2 = i1 + 1;
        if(i2 <= 3) {
            a[i2] = 0;
            int i3 = i2 + 1;
            if(i3 <= 3) {
                a[i3] = 0;
                int i4 = i3 + 1;
                if(i4 <= 3) {}
            }
        }
    }
}
```

which can be easily simplified (using constant propagation and dead code elimination) to:

**Listing 3.3.** Example A, fully unrolled and simplified

```
int* a = malloc(sizeof(int)*3);
a[0] = 0;
a[1] = 0;
a[2] = 0;
a[3] = 0; // buffer overflow is still here
```

If the loop iteration count is unknown, this transformation obviously does not preserve program semantics. For example, consider the following C code:

[1] Although our prototype is based on LLVM and uses its program representation, we use examples written in C throughout this paper for illustrative purposes.

**Listing 3.4.** Example B

```
unsigned N = ...; // some statically-unknown value
int* a = malloc(sizeof(int)*N);
for(int i = 0; i <= N; ++i) {
    a[i] = 0; // Oops, buffer overflow on last iteration
}
```

Using unroll bound of 4, this example unrolls to:

**Listing 3.5.** Example B, unrolled with a bound of 4

```
unsigned N = ...; // some statically-unknown value
int* a = malloc(sizeof(int)*N);
int i0 = 0;
if(i0 <= N) {
    a[i0] = 0;
    int i1 = i0 + 1;
    if(i1 <= N) {
        a[i1] = 0;
        int i2 = i1 + 1;
        if(i2 <= N) {
            a[i2] = 0;
            int i3 = i2 + 1;
            if(i3 <= N) {
                a[i3] = 0;
                int i4 = i3 + 1;
                if(i4 <= N) {}
            }
        }
    }
} // no buffer overflow here unless N <= 4
```

**Listing 3.6.** Example B, unrolled and simplified

```
unsigned N = ...; // some statically-unknown value
int* a = malloc(sizeof(int)*N);
a[0] = 0;
if(1 <= N) {
    a[1] = 0;
    if(2 <= N) {
        a[2] = 0;
        if(3 <= N) {
            a[3] = 0;
        }
    }
} // no buffer overflow here unless N <= 4
```

Note that we cannot further simplify the control flow of example B as we did with example A, because we do not know of any relation between N and 1, 2, 3 or 4. Any constant change to the unroll bound does not help the situation, as one could use other values (popular defaults from different analysis tools include 1, 3, 5, 10 and 100) which do not change the end result that the overflow is lost. Buffer overflows are especially sensitive to these kind of transformations, because they are often caused by “off-by-1” errors and trigger only on the last iteration.

Compiler optimizations based on loop unrolling usually have a upper limit on the unroll bound, because in many cases unrestricted loop unrolling turns to be a pessimization. Code analysis tools, on the contrary, tend to perform as full unrolling of the loop as possible even when the unroll bound is huge. Example C illustrates this kind of situation:

**Listing 3.7.** Example C

```
int** a = int_matrix_malloc(2000, 3000);
for(int i = 0; i < 2000; ++i) {
    for(int j = 0; j <= 3000; ++j) {
        a[i][j] = 0; // Oops, buffer overflow on the last iteration
    }
}
```

One can easily see that full unrolling of this example creates more than six million array accesses, i.e. the analysis complexity effectively explodes. A similar example with three nested loops could easily bring any analysis tool based on traditional loop unrolling to its knees in case it tries to do full unrolling.

## 3.3 The Context of Constraint-Based Analysis

Constraint-based analysis (e.g., bounded model checking [7]) uses an external tool (e.g., an SMT solver) to reason about programs and their properties. The analysis usually goes as follows:

1. convert the program to a logic formula
2. identify the interesting properties you want to check (e.g., error conditions)
3. ask the solver to check if these properties are feasible w.r.t. the analyzed program

Loops pose a particularly significant problem to this approach, as they cannot be easily converted to first-order logic.

Even though the original work on bounded model checking [7] employed a rather complex set of techniques to deal with loops by iteratively re-running the solver procedure, it is usually too expensive to be used in practice as every additional query to SMT solver imposes a big performance hit. That is why “unroll-and-bound” is the de-facto standard technique for dealing with loops in constraint-based analyses.

The proposed approach (see section 4) is largely based on the fact that one can introduce free (unbounded) variables into the logic formula and force the solver to infer values for them. This does lead to a performance penalty due to the additional degree of freedom every unbounded variable represents, but this is largely mitigated by the fact that the overall number of variables is greatly reduced when using “loop backstabbing”.

## 4 Enhancing the “Unroll-and-bound” Technique with Loop Backstabbing

According to the fundamental book on software testing by Boris Beizer, when you write tests for loops, you should cover at least the space of first, second, any-single-one-in-the-middle and last iterations of the loop [5]. As we’ve shown in section 3, “unroll-and-bound” covers only one half of this heuristic. In this section we present our approach to loop unrolling based on unrolling the loop both from the beginning and from the end.

### 4.1 Reaching the Last Iteration

In order to achieve our goal, we need to do two things:

– Find out the number of loop iterations[2]
– Promote all values inside the loop to their respective values on the last iteration using scalar evolution

As we piggyback ride on the existing scalar evolution implementation from LLVM, these tasks become rather trivial, as we only need to use the readily-available scalar evolution analysis. We call this process "loop backstabbing", because it nicely captures the main idea of our approach.

Let us show how loop backstabbing works on the examples from section 3. For the sake of simplicity, we will use an unrolling factor of 1 both from the beginning and from the end. Example A after backstabbing becomes:

**Listing 4.1.** Example A, unrolled and back-stabbed

```
int* a = malloc(sizeof(int)*3);
int i0 = 0;
if(i0 <= 3) { // unrolled iteration
    a[i0] = 0;
    int i1 = i0 + 1;

    if(i1 <= 3) {
        // back-stabbed iteration
        int iN = i0 + (4 - 1); // (i = i + 1) performed (4 − 1) times
        if(iN <= 3) {
            a[iN] = 0; // the overflow is here
            int iNext = iN + 1;
        }
    }
}
```

which simplifies to:

**Listing 4.2.** Example A, unrolled, back-stabbed and simplified

```
int* a = malloc(sizeof(int)*3);
a[0] = 0;
a[3] = 0; // the overflow is still here
```

Of course, we should be cautious here, as not all software bugs happen on the last iteration. Settings the unrolling factor to 1, in general, is a bad idea, and it is used here only to simplify the examples. It is also worth noting that this transformation can never introduce additional extra iterations, because the loop condition check is still present and if the loop iteration count is exceeded, the rest of the unrolled loop will be considered dead code. Therefore, loop backstabbing shows the same results as traditional loop unrolling modulo defects found.

A more interesting example to consider is example B:

**Listing 4.3.** Example B, unrolled and back-stabbed

```
unsigned N = ...; // some statically-unknown value
```

[2] In this context, number of loop iterations is a symbolic expression over loop invariants.

```
int* a = malloc(sizeof(int)*N);
int i0 = 0;
if(i0 <= N) { // unrolled iteration
    a[i0] = 0;
    int i1 = i0 + 1;

    if(i1 <= N) {
        // back-stabbed iteration
        int iN = i0 + ((N + 1) - 1); // (i = i + 1) performed ((N + 1) - 1)
            times
        if(iN <= N) {
            a[iN] = 0; // the overflow is here
            int iNext = iN + 1;
        }
    }
}
```

**Listing 4.4.** Example B, unrolled, back-stabbed and simplified

```
unsigned N = ...; // some statically-unknown value
int* a = malloc(sizeof(int)*N);
a[0] = 0;
if(1 <= N) a[N] = 0;
```

We can perform this code simplification as we know that `N` is unsigned (removing `i0 <= N` branch) and that `iN == N` (removing `iN <= N` branch), but we cannot remove `1 <= N` branch as that would break the semantics when `1 == N`.

Loop backstabbing can also be used to deal with long and nested loops. Consider example C which posed a major problem to traditional unrolling:

**Listing 4.5.** Example C, unrolled and back-stabbed

```
int** a = int_matrix_malloc(2000, 3000);
int i0 = 0;
if(i0 < 2000) {
    if(j0 <= 3000) {
        a[i0][j0] = 0;
        int j1 = j0 + 1;
        if(j1 <= 3000) {
            int jN = j0 + 3000; // (j = j + 1) performed ((3000 + 1) - 1)
                times
            if(jN <= 3000) {
                a[i0][jN] = 0;
                int jNext = jN + 1;
            }
        }
    }
    int i1 = i0 + 1;
    if(i1 < 2000) {
        int iN = i0 + 2000 - 1; // (i = i + 1) performed (2000 - 1) times
        if(iN < 2000) {
            if(j0 <= 3000) {
                a[iN][j0] = 0;
                int j1_0 = j0 + 1;
                if(j1_0 <= 3000) {
                    int jN_0 = j0 + 3000; // (j = j + 1) performed ((3000 +
                        1) - 1) times
                    if(jN_0 <= 3000) {
                        a[iN][jN_0] = 0; // and the overflow is here
                        int jNext_0 = jN_0 + 1;
                    }
                }
            }
        }
    }
}
```

**Listing 4.6.** Example C, unrolled, back-stabbed and simplified

```
int** a = int_matrix_malloc(2000, 3000);
a[0][0] = 0;
a[0][3000] = 0;
a[1999][0] = 0;
a[1999][3000] = 0; // and the overflow is still here
```

As we can see, both examples B and C no longer suffer from the problems mentioned in section 3. However, bugs can happen on any iteration, not only on the last one. Consider the following example D:

**Listing 4.7.** Example D

```
int N = ...; // some statically-unknown value
int* a = calloc(sizeof(int)*N);
if(N > 1337) {
    a[1337] = -1;
    for(int i = 0; i < N; ++i) {
        if(a[i] == -1) a[N] = 42; // overflow on iteration 1337
    }
}
```

This example proves to be challenging both to unrolling and backstabbing techniques, as the bug happens on the exact iteration that is neither first nor last. Unrolling the loop with a factor bigger than 1337 could easily solve this problem, but the erroneous iteration number can be anything and sooner or later full unrolling becomes inapplicable. Therefore, we extend our approach to solve this problem in the context of logic reasoning.

## 4.2 Loop Backstabbing in the Context of Constraint-Based Analysis

When doing constraint-based analysis, where all values are inferred by some logic reasoning tool (e.g., SMT solver), it is natural to consider the desired iteration number as yet another value to be inferred. To illustrate this idea, we need to introduce a way to control how variables are treated by the tool. In this paper we use two intrinsic functions for this purpose:

- `int __nondet__()` — a function representing some free unbounded value
- `void __assume__(int predicate)` — a function stating that `predicate` assumption is always true

In order to capture the erroneous iteration in the middle, we introduce an auxiliary free variable `iteration_count` assumed to be a valid iteration number, so that the logic tool itself can find the erroneous value. We call this process "loop mid-backstabbing".

Lets look at example A again, this time after mid-backstabbing:

**Listing 4.8.** Example A, unrolled and mid-back-stabbed

```
int* a = malloc(sizeof(int)*3);
int iteration_count = __nondet__();
int i0 = 0;
if(i0 <= 3) { // unrolled iteration
    a[i0] = 0;
    int i1 = i0 + 1;

    if(i1 <= 3) {
        // mid-back-stabbed iteration
        __assume__(iteration_count >= 0);
        __assume__(iteration_count < 4);
        int iM = i0 + iteration_count; // (i = i + 1) performed
            (iteration_ count) times
        if(iM <= 3) {
            a[iM] = 0;
            int iMNext = iM + 1;
            if(iMNext <= 3) {
                // back-stabbed iteration
                int iN = i0 + (4 - 1); // (i = i + 1) performed (4 − 1) times
                a[iN] = 0;
                int iNext = iN + 1;
            }
        }
    }
}
```

**Listing 4.9.** Example A, unrolled, mid-back-stabbed and simplified

```
int* a = malloc(sizeof(int)*3);
int iteration_count = __nondet__();
__assume__(iteration_count >= 0);
__assume__(iteration_count < 4);

a[0] = 0;
if(iteration_count <= 3) {
    a[iteration_count] = 0;
    if(iteration_count <= 4) {
        a[3] = 0;
    }
}
```

Now we can find the bug in example D the following way:

**Listing 4.10.** Example D, unrolled and back-stabbed

```
int N = ...; // some statically-unknown value
int* a = calloc(sizeof(int)*N);
int iteration_count = __nondet__();
if(N > 1337) {
    a[1337] = -1;
    int i0 = 0;
    if(i0 < N) { // unrolled iteration
        if(a[i0] == -1) a[N] = 42;
        int i1 = i0 + 1;
        if(i1 < N) {
            // mid-back-stabbed iteration
            __assume__(iteration_count >= 0);
            __assume__(iteration_count < (N - 1));
            int iM = i0 + iteration_count; // (i = i + 1) performed
                (iteration_ count) times
            if(iM < N) {
                if(a[iM] == -1) a[N] = 42;
                int iMNext = iM + 1;
                if(iMNext < N) {
                    // back-stabbed iteration
                    int iN = i0 + (N - 1); // (i = i + 1) performed (N − 1)
                        times
                    if(a[iN] == -1) a[N] = 42;
                    int iNext = iN + 1;
                }
            }
        }
    }
}
```

Assuming we have a tool that produces a counterexample for each possible erroneous case (that is exactly what most of the logic-based tools do), the only possible value that `iteration_count` can have is indeed 1337, which means that, as long as the analysis correctly handles memory reads and writes, the bug will be found. If there is no bug possible in the middle of the loop, no counterexample will be found and the logic tool can assign any possible value to `iteration_count`.

## 4.3 Handling Infinite Loops

Surprisingly enough, our backstabbing approach can be adapted to deal with infinite loops. If the loop is unconditionally infinite (i.e., does not have any exits), we can use the same backstabbing modulo the upper bound. We need yet another intrinsic function to hint that the code after the back-stabbed iteration is unreachable. We'll use `void __unreachable__()` for this (though it can be replaced with a compiler-specific unreachable marker).

Consider the following example E:

**Listing 4.11.** Example E

```
int M = ...; // some statically-unknown value
int* a = malloc(sizeof(int)*M);
int i = 0;
for( ; ; ++i) {
    a[i] = 0;
}
```

**Listing 4.12.** Example E, unrolled and back-stabbed

```
int M = ...; // some statically-unknown value
int* a = malloc(sizeof(int)*M);
int iteration_count = __nondet__();
int i0 = 0;
a[i0] = 0;
__assume__(iteration_count >= 0);
int iM = i0 + iteration_count; // (i = i + 1) performed (iteration_count)
    times
a[iM] = 0;
__unreachable__();
```

Again, if we assume we have a counterexample-based analysis tool, it is rather easy to find the bug here, as the tool can assume `iteration_count` variable has any value, for example, anything bigger than M, thus triggering the bug.

## 5 Evaluation

We have implemented loop backstabbing, mid-backstabbing and the infinite loop trick in a bounded model checker called Borealis, which currently employs the traditional “unroll-and-bound” technique, and conducted a series of experiments using an artificial test suite. This suite consists of 86 example programs of relatively small size (from 15 to 300 lines of code) that were mostly taken from SVCOMP [6] and NECLA [11] benchmarks.

Borealis has bean tweaked to use backstabbing instead of regular unrolling if it is applicable to a given loop (i.e., if the LLVM scalar evolution is powerful enough to analyze it) and to fall back to loop unrolling with a factor of 10[3] if it is not. This mode was compared against regular loop unrolling with the same constant factor of 10.

The tool (in both modes) was run on a Phenom II x4 machine with 8 GB of RAM. Both modes were run 5 times, average running time (for the whole test suite) was 9 minutes 7 seconds for the regular loop unrolling and 56 seconds for the backstabbing mode. Therefore, loop backstabbing introduces an over nine-fold increase in performance without any loss of analysis quality. We've also measured the percentage of loops that can be analyzed using the proposed approach and found out that almost 63% of all loops can be dealt with using loop backstabbing.

Besides an increase in performance, we also acquired a minor increase in analysis quality, having found no additional false positives or negatives and successfully detecting 3 new true positives and removing 2 false positives.

## 6 Related work

Our work shares most similarities with the work on loop-extended symbolic execution (LESE, [19]) and can be viewed as an application of the same underlying principle — analysis of a given loop on some arbitrary iteration — to reinforce traditional bounded model checking. Unlike LESE, however, we employ existing induction variable analyses from LLVM framework to build our back-stabbed loop representation which is used to strengthen classic bounded loop analysis.

Our approach also has some commonalities with loop under-approximation technique used in IMPULSE [13]. While this technique is more powerful and can capture inter-iteration dependencies, it uses a number of assumptions about the form of variable assignments and array accesses in the loop body which might not hold in general. Our approach, on the other hand, trades some expressive power for greater simplicity and wider applicability.

A number of other approaches also try to deal with loops using either static or dynamic approaches. For example, a technique from [10] does dynamic loop summarization and shows some interesting results, but is ill-suited for bounded model checking due to its dynamic nature. An approach from [4] focuses on the use of loop abstraction to accelerate loop termination analysis in finite state cases and, by construction, cannot analyze infinite loops.

Some recent works in software verification explore the use of interpolation to do all kinds of summarization. In [12] interpolation is used to discover strongest

[3] Factor of 10 has been used to make the analysis terminate in reasonable time.

loop invariants which, together with the iterative loop unrolling, are used to successfully analyze unbounded loops. However, as interpolation is undecidable in general, these approaches might be inapplicable performance-wise in some cases. Reachability analysis is yet another research problem that depends heavily on loop abstraction [20]. These approaches usually focus on inferring loop pre- and post-conditions w.r.t. loop reachability and delegate the task of finding bugs in the loop body to other methods.

## 7 Conclusion

In this paper we presented an approach to loop analysis based on "loop backstabbing" — instantiation of the loop body on any given iteration — that enhances traditional loop unrolling. To do that, we leverage existing scalar evolution analyses from the LLVM framework, thus avoiding the need to do complex static analyses or loop summarization techniques. Despite its simplicity, this approach allows one to analyze an arbitrary loop iteration in the middle or in the end of the loop execution without the need to do full loop unrolling.

We implemented our approach in the Borealis prototype bounded model checking tool and compared it with simple loop unrolling. The results show an over nine-fold increase in performance with a minor increase in analysis quality when using loop backstabbing. We also measured the applicability of our approach and found that, on our test suite, more than 60% of the loops can be back-stabbed.

In our future work we plan to explore the applicability of loop stabbing to techniques other than bounded model checking, evaluate different backstabbing strategies and do a more extensive evaluation.

## References

[1] Alfred V. Aho et al. *Compilers: principles, techniques & tools.* Pearson Education India, 2007.

[2] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the Saturn project. PASTE'07, pages 43–48. ACM, 2007.

[3] Marat Akhin, Mikhail Belyaev, and Vladimir Itsykson. Yet another defect detection: Combining bounded model checking and code contracts. PSSV'13, pages 1–11, 2013.

[4] Thomas Ball, Orna Kupferman, and Mooly Sagiv. Leaping loops in the presence of abstraction. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2007.

[5] Boris Beizer. *Software testing techniques.* Dreamtech Press, 2003.

[6] Dirk Beyer. Competition on software verification. TACAS'12, pages 504–524. Springer, 2012.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. TACAS'99, pages 193–207, 1999.

[8] Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Transactions on Computational Logic (TOCL)*, 2(1):1–11, 2001.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. TACAS'04, pages 168–176, 2004.

[10] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. ISSTA'11, pages 23–33, New York, NY, USA, 2011. ACM.

[11] Franjo Ivančić and Sriram Sankaranarayanan. NECLA static analysis benchmarks.

[12] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded symbolic execution for program verification. RV'11, pages 396–411, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. Under-approximating loops in c programs for fast counterexample detection. In *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 381–396. Springer Berlin Heidelberg, 2013.

[14] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. Loop summarization using abstract transformers. In *Automated Technology for Verification and Analysis*, pages 111–125. Springer, 2008.

[15] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. CGO'04, pages 75–86, 2004.

[16] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. VSTTE'12, pages 146–161, 2012.

[17] Diego Novillo. Tree ssa: A new optimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[18] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers*, pages 218–232. Springer, 2005.

[19] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. ISSTA'09, pages 225–236, New York, NY, USA, 2009. ACM.

[20] Jan Strejček and Marek Trtík. Abstracting path conditions. ISSTA'12, pages 155–165, New York, NY, USA, 2012. ACM.

[21] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. ASE'13, November 2013.