# Exploring The Relationship Between Unwind Bound and Model Checking Complexity in CBMC

**Mihai-Filip Frânculescu**[1]
**Supervisor(s): Benedikt Ahrens**[1], **Anna Lukina**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Mihai-Filip Frânculescu
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Anna Lukina, Tim Coopmans

**Abstract**

The C Bounded Model Checker (CBMC) is a widely used formal verification tool that operates directly on ANSI-C source code by bit-blasting straight-line programs into Boolean satisfiability (SAT) instances. While CBMC mitigates infinite state spaces by restricting loop iterations and recursion depths via a user-defined unwind bound ($k$) , the exact empirical relationship between this parameter and back-end solver complexity remains largely uncharacterized in existing literature. This paper systematically evaluates the impact of the maximum unwind depth on model checking complexity across three distinct software domains: array bounds, floating-point arithmetic, and heap manipulation. Utilizing metrics of execution runtime, memory footprint, and propositional variable/clause counts, we expose divergent complexity scaling behaviors. Our empirical results demonstrate that while simple non-looping floating-point operations scale under near-constant complexity , nested loops (e.g., Bubble Sort) incur a sharp quadratic complexity expansion ($O(k^2)$). Crucially, heap-intensive algorithms (e.g., In-Place Merge Sort and Singly-to-Doubly Linked List transformations) exhibit severe exponential state-space explosion, triggering solver timeouts at high unwinding thresholds. These findings challenge the intuition that model checking complexity mirrors a program's algorithmic time complexity , highlighting instead that structural loop layouts and memory-aliasing relationships dictate verification feasibility. Based on these insights, we provide concrete recommendations for iterative $k$-value scaling strategies to optimize bounded verification workflows.

# 1 Introduction

The modern world heavily relies on software systems that work consistently and predictably. As such, quality assurance of these systems is imperative. Model checking is a formal method of quality assurance which verifies that a model of a system satisfies a list of properties.

The C Bounded Model Checker (CBMC) [1] is one such model checking tool. It works directly on C code, which makes it different from classic model checkers that require writing a model of the program in a specification language separate from the code itself. Another advantage is that CBMC provides bit-precise verification by directly evaluating the raw complexity of native C data structures, pointers, and machine-sized integers without relying on data abstraction to handle them. The way CBMC simplifies computation and reduces memory and runtime is by setting a maximum depth that loops and recursion can execute, which is referred to as the k value. The manual [2] for CBMC suggests that a larger k value tends to lead to a more resource-intensive process of model checking the program, in terms of runtime and memory, but, to the extent of our knowledge, the exact details of this relationship have not been previously explored in the literature.

Our research question is "What is the impact of unwind bound on model checking complexity?". Based on our intuition, that we will explain in Section 2, we hypothesize that model checking complexity scales with time complexity of the program. We empirically study this impact on two use cases from each of the following classes of programs: array bounds, floating-point arithmetic, and heap manipulation.

The structure of the paper is as follows: Section 2 covers the background information necessary to understand this work. Section 3 shows how we chose case studies, presents our metrics, and explains our experiments to answer our questions, while Section 4 shows the results of our experiments. In Section 5 we provide analysis of the data, Section 6 discusses

ethical considerations, and finally we present conclusions and future recommendations in Section 7.

# 2 Background

In this section, we explain how CBMC works and discuss the related work in the field that informed our research question. We first describe the verification pipeline that CBMC uses to translate a C program into a decidable satisfiability problem, then discuss the theoretical guarantees and limits associated with this approach, and finally motivate the specific program classes that our experiments focus on.

## 2.1 How CBMC Works

Model checking is, in general, a formal verification technique in which the behavior of a system is checked exhaustively against a set of desired properties. Classical model checkers typically formalize the system under analysis as an abstract Kripke structure—a state-transition system in which each state is labeled with the propositions that hold within it—and verify temporal properties expressed in specification languages such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). This approach requires the engineer to construct a separate formal model of the system, distinct from its implementation.

CBMC departs from this classical approach in a way that is particularly well-suited to verifying real-world software. Rather than requiring a hand-constructed model and a temporal logic specification, CBMC operates directly on C source code, treating assertions embedded in the program itself as the specification to be checked [3, Sec. 2]. This means that a developer can express correctness properties using ordinary C `assert` statements, without needing to learn LTL, CTL, or any separate modeling language. A further practical advantage of this approach is that CBMC handles arrays, pointers, and fixed-width machine integers precisely, without abstracting away the bit-level details of their representation, which is essential when verifying low-level systems code where such details often determine correctness.

Internally, CBMC verifies a program by symbolically translating it into a Boolean satisfiability (SAT) instance, and then invoking a SAT solver to search for a satisfying assignment. If such an assignment exists, it corresponds to a concrete execution of the program that violates one of its assertions, and this assignment can be converted into a human-readable counterexample trace. If no satisfying assignment exists within the bounds explored, no violation has been found.

This translation proceeds through a sequence of well-defined stages, broadly following the approach first introduced for symbolic model checking without explicit state representations [4], and adapted by CBMC specifically for the verification of C programs:

1. The control flow of the program is first simplified into a Control Flow Graph (CFG), in which branching and looping constructs are made explicit.

2. All loops and recursive function calls in the CFG are *unwound*: each loop is replaced by a finite sequence of duplicated loop bodies, separated by conditional guards, up to a maximum number of iterations. This maximum is referred to as the *unwind bound*, and is denoted throughout this work as $k$.

3. The unwound program is converted into Static Single Assignment (SSA) form, in which every program variable is assigned a unique indexed symbol at each point it is written to, allowing the entire execution to be expressed as a conjunction of equalities.

4. The SSA representation, together with the negation of the property to be checked, is converted into a system of logical equations over bit-vectors.

5. These equations are *bit-blasted*, i.e., translated into an equivalent Boolean formula over individual bits, which is the input format required by a SAT solver.

6. An efficient SAT solver is invoked on the resulting Boolean formula to determine whether a satisfying assignment exists.

7. If a satisfying assignment is found, it is mapped back to the original program variables and control flow, yielding a concrete counterexample trace that violates the property under test.

The role of the unwind bound $k$ in this pipeline is central to the present work. Because a SAT instance must be finite, CBMC cannot reason about loops or recursive calls of unbounded depth directly; it can only verify the program up to $k$ iterations or recursive calls of any given loop or function. If $k$ is set too low to fully traverse a loop's intended behavior, CBMC will not silently produce an unsound result; instead, it emits an explicit *unwinding assertion failure*, indicating that the chosen bound was insufficient to determine whether the property holds. This makes the choice of $k$ a critical and explicit parameter of every verification run, and is the central subject of the experiments described in Section 3.

## A Worked Example

To make the verification pipeline described above concrete, consider the following small C function, which computes the sum of the elements of a fixed-size global array:

```
int array[10];

int sum() {
    unsigned i, sum;

    sum = 0;
    for (i = 0; i < 10; i++)
        sum += array[i];

    return sum;
}
```

This function contains a single loop, controlled by the variable `i`, which iterates exactly ten times over the array. Before this program can be converted into a SAT instance, CBMC must first unwind the `for` loop into a finite, loop-free sequence of statements. Given an unwind bound of $k = 10$, the loop is unrolled into ten sequential copies of its body, each guarded by the corresponding loop condition. Conceptually, this unwinding produces a program equivalent to the following:

```
int array[10];

int sum() {
    unsigned i, sum;

    sum = 0;
    i = 0;

    if (i < 10) {
        sum += array[i]; i++;
        if (i < 10) {
            sum += array[i]; i++;
            if (i < 10) {
                sum += array[i]; i++;
                /* ... nested up to 10 levels deep ... */
                if (i < 10) {
                    sum += array[i]; i++;

                    if (i < 10) assert(0); /* unwinding assertion */
                }
            }
        }
    }

    return sum;
}
```

Each unwound copy of the loop body is nested inside the guard of the previous one, since the $j$-th iteration is only reachable if the loop condition was still satisfied after the $(j-1)$-th iteration; a flat sequence of independent `if` statements would incorrectly allow a later iteration to execute even when an earlier guard had already failed. The innermost guard above is CBMC's *unwinding assertion*: if, after $k$ unwindings, the loop condition could still be true, this indicates that $k$ was not large enough to fully explore the loop, and CBMC reports an unwinding assertion failure rather than silently producing an unsound result. In this example, since the loop is known to execute exactly ten times, any $k \geq 10$ is sufficient to fully unwind it; choosing $k = 5$, for instance, would cause the unwinding assertion to fail, since the loop condition `i < 10` would still be satisfiable after only five iterations.

Once the loop has been fully unwound, CBMC converts the resulting straight-line program into Static Single Assignment (SSA) form, in which each assignment to a variable introduces a fresh indexed symbol. For instance, the repeated updates to `sum` and `i` would be rewritten using fresh symbols at each iteration, such as $sum_1, sum_2, \ldots, sum_{10}$ and $i_1, i_2, \ldots, i_{10}$, with each equation of the form $sum_j = sum_{j-1} + array[i_{j-1}]$. This SSA representation is then translated into a system of bit-vector equations, bit-blasted into a Boolean formula, and passed to a SAT solver. A satisfying assignment to the negation of a property—for example, an assertion that the returned sum never exceeds some fixed bound—would correspond to a concrete counterexample input violating that property; the absence of a satisfying assignment, conversely, confirms that no such violation exists within the unwound program.

This example, although simple, illustrates precisely why the unwind bound $k$ is con-

sequential: each additional unwound iteration introduces further SSA equations and, consequently, further variables and clauses into the final SAT instance, directly affecting the cost of solving it. The experiments in Section 3 systematically examine this effect across programs of varying structural complexity.

## 2.2 Soundness, Completeness, and the Unwind Bound

The reliability of any model checking technique rests on two complementary guarantees: *soundness*, meaning that any property the tool reports as verified actually holds, and *completeness*, meaning that the tool will eventually detect any property that does not hold. Bounded model checking, as implemented in CBMC, is sound but only conditionally complete. Any counterexample CBMC reports corresponds to a genuine execution of the program; however, if no counterexample is found at a given $k$, this only guarantees the absence of violations reachable within $k$ unwindings. It does not, in general, guarantee the absence of violations that would only manifest at a greater depth.

Completeness can be recovered in principle by unwinding each loop up to its *completeness threshold*: a bound beyond which no new program states can be reached, since the loop's behavior past that point is necessarily a repetition of previously explored states. In practice, however, this threshold is frequently far too large to be computed or unwound within feasible time or memory, particularly for loops whose iteration count depends on input data rather than a fixed constant. This is closely related to the broader phenomenon of *state space explosion*, in which the number of distinct states or execution paths through a program grows exponentially with the depth of exploration, rapidly overwhelming the capacity of the underlying solver.

An alternative technique that sidesteps the need for impractically large unwind bounds is *k-induction*, in which correctness is established not by exhaustively unwinding a loop, but by proving a base case for the first few iterations and an inductive step showing that if the property holds for any $k$ consecutive states, it must also hold for the following state. When applicable, k-induction allows a property to be proven for a loop of unbounded length without unwinding it indefinitely. This work, however, is concerned specifically with the behavior of CBMC under direct bounded unwinding via the $k$ parameter, rather than such inductive extensions.

## 2.3 Strengths and Limitations Across Program Classes

The practical effectiveness of CBMC's bounded unwinding approach is known to vary substantially depending on the structure of the program being verified. The literature broadly identifies certain classes of programs for which CBMC performs reliably and efficiently, and other classes for which verification becomes considerably more costly as the unwind bound increases. The present study focuses on three such classes.

**Array bounds and buffer safety.** Programs whose correctness properties concern indexing within fixed-size arrays or buffers are a well-established strength of CBMC. Because array sizes and memory layouts are handled precisely rather than abstractly, CBMC is particularly effective at detecting out-of-bounds accesses and verifying index-related invariants in algorithms operating over arrays [5].

**Floating-point arithmetic.** In contrast, the verification of programs involving floating-point computation has historically posed greater difficulty for bounded model checkers. The bit-level encoding of IEEE 754 floating-point operations introduces substantial additional

complexity into the generated SAT instance, and prior work has proposed mixed abstraction techniques specifically to mitigate the cost of reasoning about floating-point arithmetic within CBMC [6]. This makes floating-point arithmetic a representative class for examining how the unwind bound interacts with an intrinsically more expensive encoding.

**Heap manipulation.** Programs that dynamically allocate and manipulate heap-based data structures, such as linked lists, present a further class of difficulty, as the symbolic representation of the heap must account for an increasing number of possible memory layouts and aliasing relationships as the program is unwound to greater depths [7]. This makes heap manipulation a useful counterpart to floating-point arithmetic for studying how structural, rather than purely arithmetic, complexity scales with $k$.

It is worth noting that concurrency is a further class of programs for which CBMC has seen substantial development over the past two decades, from early work on bounding the number of context switches considered during verification [8], to dedicated concurrency preprocessing tools such as CSeq [9], to techniques for reducing the number of interleaving instances that must be explored [10]. Despite this progress, concurrency verification introduces an additional dimension to the unwind bound, since $k$ simultaneously governs both loop unwinding depth and the depth of thread interleaving considered, making it difficult to isolate the effect of $k$ on complexity independently of interleaving effects. For this reason, concurrency is excluded from the experiments in this work, and we restrict our attention to array bounds, floating-point arithmetic, and heap manipulation, where the unwind bound can be studied in a more controlled and interpretable manner.

These three classes directly motivate the case study selection described in Section 3, where representative programs from each category are subjected to a systematic sweep of unwind bounds in order to empirically characterize the relationship between $k$ and verification complexity.

# 3 Methodology

In this section, we will explain the experimental setup that we use to answer our research question. First, we cover case study selection, afterwards we discuss the metrics used to evaluate the complexity of model checking, and finally we describe the experimental setup used.

## 3.1 Case Study Selection

Throughout the papers we evaluated, most examples were pretty small in scope, but thoroughly examined. Following this standard, we decided to choose two use cases for each class of programs defined in Section 2. For Array Bounds, we choose Bubble Sort and Binary Search from [5], as they are standard algorithms in computer science and they have enough complexity to show interesting results when scaling up. For the other two categories, we turned to the Competition on Software Verification (SV-Comp) [11], which is well renowned in the field as the largest software verification competition. It provides a large repository of programs that are used to benchmark model checkers, and they have programs for floating-point arithmetic, as well as heap manipulation. We pick Float to Double conversion and Inverse Square as our programs for the Floating-Point Arithmetic section, due to their simplicity. Finally, for Heap Manipulation, we choose Singly Linked List to Doubly Linked List transformation (referred to as SLL to DLL) and In Place Merge Sort, as these are easy to understand and explain, but complex enough to introduce difficulty.

Small modifications were necessary for some of the use cases, such as generalizing the bubble sort algorithm to run on arbitrary arrays. As such, we provide our versions for each of the use cases in [12].

### 3.2 Metrics

The metrics we chose are the elapsed time, memory used, and the size of the generated SAT instance, measured by the number of variables and clauses. The first paper[4]

### 3.3 Experimental Setup

In order to investigate the influence of k on verification complexity, we have conducted a series of experiments across three categories: array bounds, floating-point arithmetic, and heap manipulation. We selected these categories to reflect both the known strengths and known limitations of CBMC as documented in the literature, allowing for a direct comparison of how k impacts verification feasibility across structurally different classes of programs.

All experiments will be run on a single fixed machine running Windows 11, with an AMD Ryzen 7 CPU and 16 GB of RAM. We used version CBMC version 6.8.0 within Windows Subsystem for Linux.

## 4 Results

In this section, we present the results of our experiments with respect to varying unwinding bounds ($k$). As mentioned in Section 2 and Section 3, we have chosen programs in three classes: Array Bounds, Floating-Point Arithmetic and Heap Manipulation.

### 4.1 Array Bounds

Bubble Sort is a simple comparison-based sorting algorithm that operates by iteratively stepping through an array, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until no more swaps are required. Over successive passes, larger elements "bubble up" to their correct positions at the end of the structure. Because it relies on nested iterations tracking every possible inversion, verifying unoptimized implementations dynamically yields a sharp quadratic complexity expansion ($O(k^2)$).

Binary Search is an efficient O(log n) search algorithm designed for sorted arrays. It operates via a divide-and-conquer strategy, continually halving the search space by comparing the target value to the middle element of the current array bounds. If the target is smaller, the upper half is discarded; if larger, the lower half is discarded. For bounded model checkers, verifying binary search evaluates logarithmic execution depths, resulting in highly stable performance scaling across increasing unwinding bounds.

We show the versions of these results without optimization options turned on, to see the representative impact of k value on model checking complexity.

| $k$ | Outcome | Time (s) | RAM (MB) | Variables | Clauses |
|---|---|---|---|---|---|
| 5 | FAILURE | 0.03 | 16.80 | 10,600 | 36,878 |
| 10 | FAILURE | 0.06 | 22.02 | 17,595 | 65,048 |
| 20 | FAILURE | 0.09 | 29.11 | 31,585 | 121,763 |
| 50 | FAILURE | 0.24 | 54.45 | 73,555 | 294,908 |
| 100 | FAILURE | 0.51 | 100.34 | 143,505 | 593,483 |

Table 1: Impact of Unwinding Bound ($k$) on Model Checking Complexity for binary search

Table 2: Impact of Unwinding Bound ($k$) on Model Checking Complexity for Bubble Sort

| $k$ | Outcome | Time (s) | RAM (MB) | Variables | Clauses |
|---|---|---|---|---|---|
| 1 | SUCCESS | 0.04 | 12.27 | 1,066 | 127 |
| 2 | SUCCESS | 0.03 | 13.03 | 2,857 | 2,051 |
| 5 | SUCCESS | 0.07 | 18.81 | 19,670 | 20,277 |
| 10 | SUCCESS | 0.22 | 45.13 | 93,717 | 128,351 |
| 20 | SUCCESS | 0.84 | 141.76 | 322,178 | 638,709 |
| 50 | SUCCESS | 6.47 | 845.04 | 1,808,615 | 4,543,771 |
| 100 | SUCCESS | 32.58 | 3,393.15 | 6,960,936 | 19,684,749 |

## 4.2 Floating-Point Arithmetic

Floating-point type promotion (converting a 32-bit single-precision 'float' to a 64-bit double-precision 'double') involves re-encoding the structural components defined by the IEEE 754 standard. The hardware extracts the single sign bit, expands the 8-bit biased exponent to an 11-bit biased exponent, and pads the trailing 23-bit significand (mantissa) with zeros to fill the wider 52-bit slot. In formal verification, tracking these precise bit-shifts and layout mutations requires bit-blasting the floating-point logic to prevent precision mismatches or rounding bugs.

| $k$ | Outcome | Time (s) | RAM (MB) | Variables | Clauses |
|---|---|---|---|---|---|
| 5 | SUCCESS | 0.01 | 12.42 | 0 | 0 |
| 10 | SUCCESS | 0.01 | 12.56 | 0 | 0 |
| 20 | SUCCESS | 0.01 | 12.48 | 0 | 0 |
| 50 | SUCCESS | 0.01 | 12.44 | 0 | 0 |
| 100 | SUCCESS | 0.00 | 12.60 | 0 | 0 |

Table 3: Impact of Unwinding Bound ($k$) on Model Checking Complexity for float double

The Fast Inverse Square Root algorithm (famous from Quake III) computes 1 / sqrt(x) rapidly using low-level bit manipulation instead of expensive floating-point division. It achieves this by taking a floating-point value, casting its raw bits directly into a 32-bit integer, and performing an integer subtraction from a specific geometric magic constant (0x5F3759DF) to compute a remarkably accurate initial guess. This approximation is subsequently refined using a single iteration of Newton's method.

| $k$ | Outcome | Time (s) | RAM (MB) | Variables | Clauses |
|---|---|---|---|---|---|
| 5 | SUCCESS | 0.02 | 14.19 | 3,309 | 15,324 |
| 10 | SUCCESS | 0.02 | 14.16 | 3,309 | 15,324 |
| 20 | SUCCESS | 0.02 | 14.35 | 3,309 | 15,324 |
| 50 | SUCCESS | 0.02 | 14.19 | 3,309 | 15,324 |
| 100 | SUCCESS | 0.02 | 14.20 | 3,309 | 15,324 |

Table 4: Impact of Unwinding Bound ($k$) on Model Checking Complexity for inv square

### 4.3 Heap Manipulation

In-Place Merge Sort retains the divide-and-conquer architecture of standard merge sort-recursively splitting the array into halves, sorting them, and combining them-but eliminates the auxiliary O(n) memory footprint. The merging phase is executed directly within the original array boundaries using pointer shifts, index manipulation, or internal block-swapping mechanics to slide elements into their correct sequence. Because it avoids heap reallocation, it has a highly predictable memory footprint under formal verification, though the shifting logic heavily complicates data-dependency tracking.

Table 5: Impact of Unwinding Bound ($k$) on Model Checking Complexity for In Place Merge Sort

| $k$ | Outcome | Time (s) | RAM (MB) | Variables | Clauses |
|---|---|---|---|---|---|
| 5 | SUCCESS | 0.59 | 48.80 | 74,829 | 273,610 |
| 10 | SUCCESS | 3.95 | 170.05 | 304,962 | 1,256,265 |
| 20 | SUCCESS | 41.88 | 667.91 | 1,231,491 | 5,404,100 |
| 50 | SUCCESS | 3081.84 | 4,561.50 | 7,740,220 | 36,616,727 |
| 100 | TIMEOUT | — | — | — | — |

Transforming a Singly Linked List into a Doubly Linked List requires augmenting the underlying node topology to support bidirectional traversal. The algorithm iterates through the sequential structure; at each node, it tracks the memory address of the current node and injects it into the newly introduced 'prev' pointer field of the succeeding node. For model checkers, verifying this structure requires robust pointer aliasing analysis to ensure that no cyclical memory leaks, null dereferences, or dangling references occur during pointer assignment.

Table 6: Impact of Unwinding Bound ($k$) on Model Checking Complexity for SLL to DLL

| $k$ | Outcome | Time (s) | RAM (MB) | Variables | Clauses |
|---|---|---|---|---|---|
| 5 | SUCCESS | 0.65 | 48.60 | 74,829 | 273,610 |
| 10 | SUCCESS | 4.08 | 171.45 | 304,962 | 1,256,265 |
| 20 | SUCCESS | 44.17 | 668.02 | 1,231,491 | 5,404,100 |
| 50 | TIMEOUT | 356.02 | 4,216.95 | — | — |

# 5 Discussion

In this section, we analyze and interpret the results of our experiments from Section 4. All of our metrics show complexity in different ways, but generally show the same trends for individual program classes.

## 5.1 Array Bounds

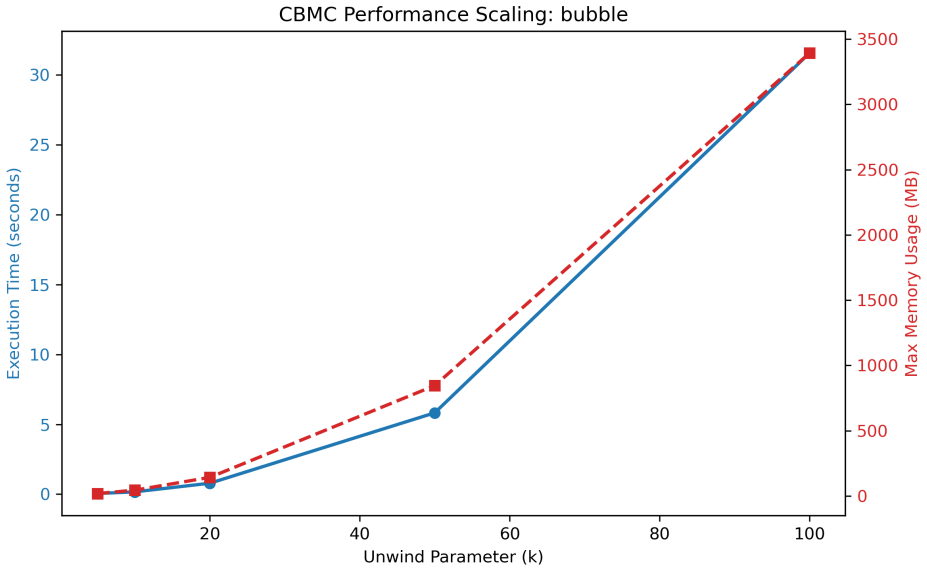


Figure 1: Solver time and memory vs k value for bubble sort.

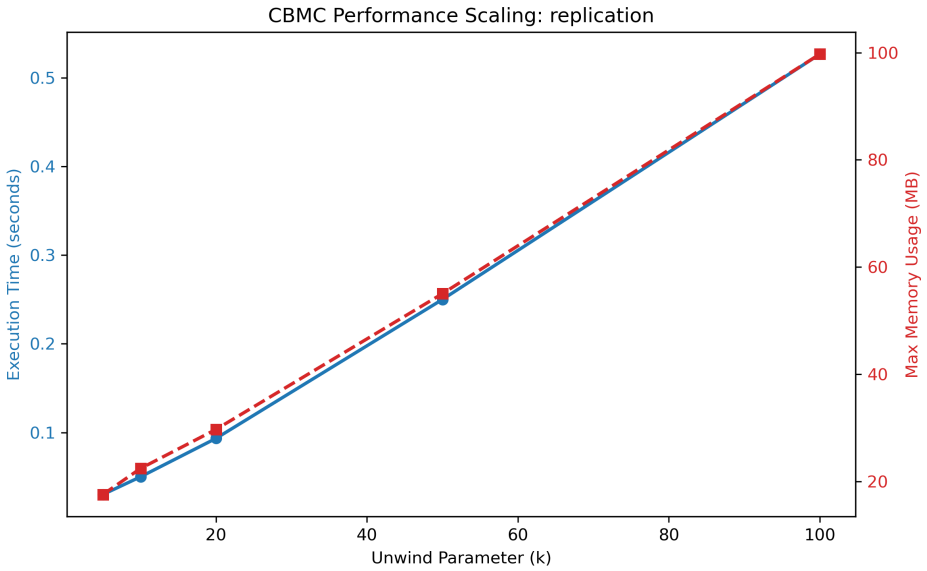


Figure 2: Solver time and memory vs k value for binary search.

For the Array Bounds class of problems, our results in Figure 1 suggest a quadratic increase in model checking complexity occurs when the program contains nested loops, as is the case for Bubble Sort. This aligns with our intuition about CBMC loop unwinding, that is, nested loops increase quadratically with k. What is worthy of note is that the operations within the loop are quite simple, so they do not introduce any more time or memory usage than expected. Conversely, for Binary Search, which has logarithmic time complexity, we see a linear increase in complexity in Figure 2, which is somewhat unexpected. After further examination, this result highlights that the complexity of model checking does not scale with time complexity of the program, but rather with loop size and count. When the operations within the loop are small comparisons and few arithmetic operations that are not too complex to transform into a SAT instance, then all the complexity is added by more loops, or nested loops, which comes through in this case. We have a single for loop that is unwound in its entirety by CBMC, leading to the linear scaling with k.

## 5.2 Floating Point Arithmetic

For the Floating Point Arithmetic class, our complexity metrics in Tables 3 and 4 remain generally constant across all tested k values. At first glance, this might suggest that floating point operations are inherently too simple to meaningfully affect model checking complexity as k increases. After further examination, however, this result is better explained by the structure of the chosen programs rather than by floating point arithmetic itself: neither Float to Double Conversion nor Inverse Square contains a loop whose iteration count is governed by k, so increasing the unwind bound has no effect on the size of the unwound program or the resulting SAT instance. This indicates that, for floating point arithmetic specifically, k only becomes a relevant factor in programs where floating point computations are themselves performed within a loop, such as iterative numerical approximations, and

that constant complexity in this class should be attributed to the absence of such loops rather than to the simplicity of floating point encoding in general.

### 5.3 Heap Manipulation

For the Heap Manipulation class we see exponential growth for both of the use cases in Figures 3 and 4, even reaching a TIMEOUT for some of the higher k values that we tested with. After further examination, this result suggests that the impact of k on complexity is not determined solely by the number or nesting of loops, but also by the cost of the operations performed within each unwound iteration. In both In Place Merge Sort and SLL to DLL, each loop iteration involves dynamic memory access and pointer manipulation, which requires CBMC to symbolically track an increasing number of possible heap layouts and aliasing relationships as k grows. This indicates that the difficulty of individual operations within a loop body is itself a contributing factor to model checking complexity, distinct from and compounding the effect of loop count or nesting observed in the Array Bounds class.

### 5.4 General Remarks

Comparing all of our results together, we suggest that programs that are not dependent on loops tend to model check in constant time, regardless of k value, while programs with loops, or programs where the operations are more complex are very sensitive to k value. As such, we recommend a strategy of increasing k value slowly while model checking.

## 6 Responsible Research

This section discusses the ethical implications of this work, namely the source of the code examples for the experiments, the reproducibility of the experiments, the usage of AI for this work, and the potential implications that this research has.

### 6.1 Reproducibility

As mentioned above, all code is open source, taken from other papers [5] or online repositories [13]. Because all of the code is freely available and the details of the experimentation are presented in Section 3, the experiments in this paper can easily be reproduced. To avoid biased interpretation of the results, we used multiple optimization settings for CBMC and ran the experiments multiple times to avoid outliers due to background processes.

### 6.2 AI usage

For this research project, we used generative AI for an initial understanding of CBMC, by asking about its' features, and generating some toy examples to test the tool on. For writing the paper itself, we only used generative AI for feedback on the writing style of some paragraphs, and we also used AI for help with LaTeX syntax. All AI responses were evaluated and rephrased before possible inclusion in the text.

### 6.3 Implications

Finally, while we believe that this paper does not raise any risk for the industry or scientific community, having empirical evidence for the relationship between k and model

checking complexity can be useful for informing future work on the tool and for deciding whether this tool is useful for a reader of this paper.

# 7 Conclusions and Future Work

This work set out to investigate the relationship between CBMC's unwind bound $k$ and the complexity of model checking, measured in terms of verification time, memory usage, and the size of the generated SAT instance. Across six benchmark programs spanning three program classes— array bounds and buffer safety, floating-point arithmetic, and heap manipulation—we found that this relationship is far from uniform. Programs involving nested loops or heap-based data structures, such as bubble sort, in-place merge sort, and the singly-to-doubly linked list transformation, exhibited exponential growth in runtime, memory, and SAT instance size as $k$ increased, with several runs becoming intractable within practical time limits. Programs whose complexity scales more moderately with input size, such as binary search, showed a more gradual, near-linear increase. Notably, our floating-point benchmarks showed essentially constant complexity across all tested values of $k$, indicating that the cost of floating-point arithmetic in CBMC is not, by itself, sensitive to the unwind bound, and instead depends on whether the program contains loops whose unwinding is governed by $k$ in the first place.

The main contribution of this work is the empirical evidence supporting the exponential impact of larger unwind bounds on CBMC runtime, memory usage, and generated SAT instance size. Based on the results discussed in Section 4, we believe that a higher maximum unwind depth has a negative effect in programs with many nested loops or complex operations performed in each loop. As such, we recommend slowly increasing the $k$ value while performing the method mentioned in Section 2 for a faster iterative process of validating the correctness of the software.

The main limitations of this work are the size and number of case studies chosen for the experimentation. More examples could offer more confidence in the conclusion, and larger case studies might show a different relationship between $k$ and complexity that cannot be captured at a smaller size. As such, we recommend more experimentation on larger case studies, possibly taken from production code, as this would also offer a more representative sample for real usage of the tool. Furthermore, automatic recommendation of $k$ value based on heuristics is an interesting direction of research that can be followed in the future.

# References


[1] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (K. Jensen and A. Podelski, eds.), vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[2] D. Kroening and M. Tautschnig, "Cbmc reference manual." `https://diffblue.github.io/cbmc/`, 2024. Accessed: 2026-06-07.

[3] D. Kroening, P. Schrammel, and M. Tautschnig, "Cbmc: The c bounded model checker," 2023.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," 1 1999.

[5] H. Collavizza, M. Rueher, and P. Van Hentenryck, "Comparison between cpbpv, esc/-java, cbmc, blast, eureka and why for bounded program verification," *arXiv preprint arXiv:0808.1508*, 2008.

[6] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *2009 Formal Methods in Computer-Aided Design*, pp. 69–76, IEEE, 2009.

[7] H. Lauko, L. Korenčik, and P. Ročkai, "Verification of programs sensitive to heap layout," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–27, 2022.

[8] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, vol. 3576 of *Lecture Notes in Computer Science*, pp. 82–97, Springer, 2005.

[9] B. Fischer, O. Inverso, and G. Parlato, "Cseq: A concurrency pre-processor for sequential c verification tools," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 710–713, IEEE, 2013.

[10] N. Lam, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato, "Parallel bug-finding in concurrent programs via reduced interleaving instances," 09 2017.

[11] D. Beyer and J. Strejček, "Evaluating software verifiers for C, Java, and SV-LIB (report on SV-COMP 2026)," in *Proceedings of the 32nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 16506 of *Lecture Notes in Computer Science*, pp. 461–502, Springer, 2026.

[12] M. F. Franculescu, "Research-project-cbmc." `https://github.com/CevaNume/Research-Project-CBMC`, 2026. Accessed: 2026-06-07.

[13] SoSy-Lab, "Software verification benchmark collection (sv-benchmarks)." `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`, 2026. Accessed: 2026-06-21.

# A Examples of AI Usage

This appendix documents three representative prompts submitted to a generative AI assistant during the course of this project, along with the responses received. These examples illustrate the nature of AI assistance used, as described in Section 6.2 (AI Usage).

### Prompt 1: Generating a toy example for CBMC

**Prompt**

Please give me a toy example to test CBMC with.

**Response**

A simple and illustrative starting point is a small loop that sums the elements of a fixed-size array:

```
#include <assert.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int sum = 0;
    int i = 0;

    while (i < 5) {
        sum += arr[i];
        i++;
    }

    assert(sum == 15);
    return 0;
}
```

This example is well suited as an introductory test case because it is small enough to trace by hand, while still exercising the core mechanism CBMC relies on: loop unwinding. Running `CBMC` on this file with a sufficiently high `-unwind` bound (at least 5) allows the loop to be fully explored, and the assertion will hold. Setting the bound too low (e.g. `-unwind 3`) causes CBMC to report an unwinding assertion failure, illustrating concretely why the unwind bound parameter matters.

## Prompt 2: LaTeX table formatting

**Prompt**

How to make a table in latex with 5 columns and 5 rows.

**Response**

A 5-column, 5-row table (including a header row) can be created using the standard `table` and `tabular` environments as follows:

```
\begin{table}[h]
\centering
\caption{Example table with 5 columns and 5 rows}
\label{tab:example}
\begin{tabular}{ccccc}
\toprule
Col 1 & Col 2 & Col 3 & Col 4 & Col 5 \\
\midrule
A1 & B1 & C1 & D1 & E1 \\
A2 & B2 & C2 & D2 & E2 \\
A3 & B3 & C3 & D3 & E3 \\
A4 & B4 & C4 & D4 & E4 \\
\bottomrule
\end{tabular}
\end{table}
```

The `ccccc` argument specifies five centered columns. Using `booktabs` (via `\usepackage{booktabs}`) for `\toprule`, `\midrule`, and `\bottomrule` produces cleaner horizontal lines than the default `\hline`.

### Prompt 3: Rephrasing a sentence for clarity

**Prompt**

Please rephrase this sentence: For the Floating Point Arithmetic class, our complexity metrics are generally constant, suggesting that floating point operations are not complex enough to meaningfully affect model checking complexity when increasing k value.

**Response**

A clearer phrasing of this sentence is:

> *For the Floating-Point Arithmetic class, our complexity metrics remain largely constant across all tested unwind bounds, suggesting that floating-point operations alone do not introduce sufficient complexity to meaningfully affect verification cost as the unwind bound increases.*

This revision improves clarity by separating the observation (metrics remain constant) from the interpretation (floating-point operations alone are not the driver of complexity), and replaces the informal phrase "not complex enough" with more precise academic language.