Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

# Automated Loop Detection and Iteration Count Analysis in Binary Code

**Hayk Aslanyan**

hayk.aslanyan@rau.am

Russian-Armenian University

**Garnik Khroyan**

Russian-Armenian University

**Shake Hakobyan**

Russian-Armenian University

**Hripsime Hovhannisyan**

Russian-Armenian University

**Research Article**

**Keywords:** static analysis, binary code analysis, loop detection, iteration count

**Posted Date:** July 8th, 2026

**DOI:** https://doi.org/10.21203/rs.3.rs-10206957/v1

**License:** This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

**Additional Declarations:** No competing interests reported.

# Automated Loop Detection and Iteration Count Analysis in Binary Code

Hayk Aslanyan[1*], Garnik Khroyan[1], Shake Hakobyan[1], Hripsime Hovhannisyan[1]

[1]Center of Advanced Software Technologies, Russian-Armenian University, 123 Hovsep Emin street, Yerevan, 0051, Armenia.

*Corresponding author(s). E-mail(s): hayk.aslanyan@rau.am;
Contributing authors: khroyan.garnik@rau.am;
shake.hakobyan@rau.am; hripsime.hovhannisyan@rau.am;

**Abstract**

Accurate analysis of loops in programs is essential for performance prediction and vulnerability detection. However, aggressive compiler optimizations often transform source code loops, making direct analysis of compiled binaries necessary. A fundamental challenge in this context is the precise determination of loop iteration counts, which are used for worst-case execution time (WCET) analysis and detecting potential infinite loops. This paper presents an automated and scalable method for identifying natural loops, iteration variables, and corresponding iteration counts in binary code. The proposed method combines interprocedural static analysis with control-flow and data-dependency analysis to track variable definitions across instructions, function calls, and returns. The method consists of three stages. First, it detects natural loops and constructs use-def chains over binary instructions. Second, it identifies iteration variables by analyzing loop-controlling predicates and the variables influencing them. Third, it determines exact iteration counts where statically decidable by analyzing iteration variable update patterns and termination conditions. Experimental evaluation on real-world software and a benchmark suite demonstrates high accuracy and precision in loop detection and iteration count estimation, outperforming existing approaches in coverage while maintaining correctness. The developed tool achieves 100% precision and 60.2% recall for the determination of loop iteration counts on the Mälardalen WCET benchmark suite. Additionally, the tool successfully handles binaries of tens of megabytes in size. Our experiments show that the complete analysis, including disassembly and all analysis stages, processes 1 MB of AArch64 binary code in under 20 seconds, indicating practical scalability.

**Keywords:** static analysis, binary code analysis, loop detection, iteration count

# 1 Introduction

Loops are a fundamental component of program structure and are particularly important in program analysis because they often dominate program execution time [1]. Compilers frequently transform or eliminate source-level loops through optimization [2, 3], meaning the loops in the source code may differ significantly from those in the compiled binary [4]. This makes binary analysis particularly valuable for understanding the runtime behavior of optimized programs because it examines the actual instructions that are executed. Such analysis is essential for various applications, including security vulnerability detection [5, 6], reverse engineering and decompilation [7], performance analysis [8], worst-case execution time (WCET) analysis for real-time systems [9, 10], and malware analysis [11].

Despite decades of research on loop analysis, most existing approaches have been designed for source code or specialized intermediate representations that retain high-level program information. As a result, these methods cannot effectively analyze optimized binaries, where compiler optimizations, missing type information, and interprocedural data dependencies obscure loop boundaries, making the reconstruction of loop control flow and iteration behavior non-trivial. These limitations create the need for a new binary-level technique that can analyze machine instructions directly while providing automation and accuracy.

Analyzing loops in binary code requires understanding their underlying structure. In this work, we focus on natural loops [12], which are characterized by having a single entry point (the header). Non-natural loops, which have multiple entry points, typically arise from aggressive compiler optimizations. Our experiments show that they account for approximately 10% of all loops.

Among the various aspects of natural loop analysis, an essential one is estimating loop iteration counts—the total number of times the loop header is executed [10, 13]. Accurate iteration count estimation enables performance prediction, helping to detect potential infinite or unbounded loops [14, 15]. It is used for WCET analysis in real-time systems [9, 16], as well as energy consumption analysis [17, 18]. A key step in determining loop iteration counts is identifying iteration variables. These are counters that update their value in each loop iteration based on their previous value and determine the loop's exit condition.

This paper introduces a method for identifying natural loops, their iteration variables, and iteration counts in binary code through interprocedural static analysis. The method traces data dependencies both within functions and across function calls and returns, analyzes how loop counters propagate through procedure boundaries, and computes loop iteration counts. The approach is fully automated and scalable to binaries of dozens of megabytes (corresponding to millions of lines of source code).

The rest of this paper is organized as follows: Section 2 presents the overall method and workflow of the analysis. Section 3 provides an overview of the relevant literature, including techniques for detecting loop iteration variables and iteration bounds. In Section 4, we describe the intermediate representations used for binary code and discuss the solutions for detecting natural loops, their nested structures, and data dependency analysis needed for the following analysis stages. Section 5 describes our approach for identifying iteration variables. Section 6 focuses on the method of detecting iteration counts. Section 7 discusses the implementation details and system architecture of our approach. Evaluation results of the designed tool are discussed and presented in Section 8. Finally, Section 9 concludes the paper and discusses future research directions.

## 2 Method Overview

Our method consists of a sequence of analysis stages that detect natural loops, their nesting hierarchy, iteration variables, and corresponding iteration counts from binary code (Figure 1). The process begins with binary disassembly, which extracts low-level structural artifacts including assembly instructions, operand read/write effects, basic blocks, and control flow graphs (CFGs).

It is followed by two independent stages that analyze control flow and data dependency, respectively. The control flow analysis stage identifies natural loops by detecting headers and back edges [12] in CFGs, producing collections of basic blocks that represent individual loops and their nesting hierarchy. Data dependency analysis constructs use-def chains [12] by tracking how variable definitions reach their corresponding uses throughout the program.

The method identifies iteration variables using a multi-stage algorithm. It first identifies conditional branch instructions that control loop execution. It then traverses use-def chains to identify variables that influence these branch conditions. For loops that satisfy predefined conditions (single iteration variable, single loop condition, linear update of the iteration variable), the algorithm then determines precise iteration counts by emulating instructions that change the iteration variable, constructing and solving equations.

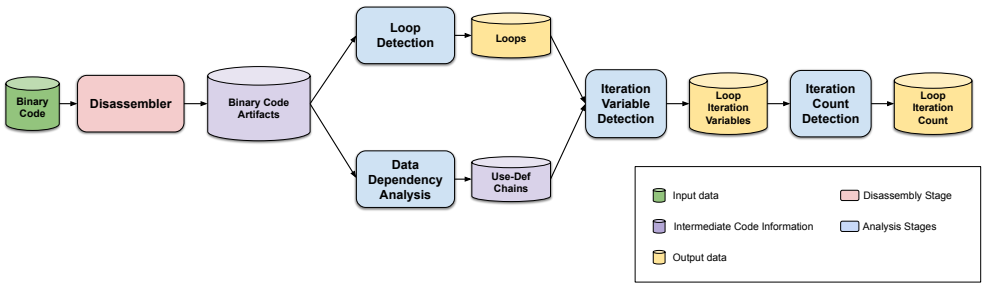



**Fig. 1** Overview of the proposed method.

# 3 Related Work

Precise analysis of program loops plays two primary functions in computer science: assisting compilers in optimizing programs and allowing formal verification of safety-critical systems. These dual directions have provided distinct but complementary analysis techniques. We organized existing research into three categories—algebraic approaches, logic-based verification methods, and integrated frameworks—examining their capabilities and their limitations at the binary level.

Induction variables (IVs) are variables whose values form arithmetic or geometric progressions with loop iterations. Wolfe's work [19] extended this concept beyond simple linear IVs to Monotonic Variables—variables that consistently increase or decrease without necessarily following arithmetic patterns. The work classified loop-variant variables into linear IVs, periodic, wrap-around, and monotonic categories, providing a foundation for iteration variable identification. Building on these foundations, modern compilers like LLVM [20] and GCC [21] employ the Scalar Evolution (SCEV) framework, which models variable progression through Chains of Recurrences (CR) algebra—first proposed by Bachmann (1994) [22] and refined by Van Engelen (2001) [23]. SCEV also computes loop trip counts by modeling the induction variable as a symbolic recurrence and solving for when the exit condition becomes true, producing exact symbolic bounds (e.g., "n iterations" for simple counting loops) that enable precise loop transformations. SCEV operates exclusively on source code with explicit SSA-form and rich type information—at the binary level, registers are reused across logical variables and compiler optimizations restructure control flow beyond recognition.

Verification-focused methods prioritize soundness over precision. Abstract Execution (AE), implemented in the SWEET tool [16, 24], combines symbolic execution (tracking symbolic values instead of concrete ones) and abstract interpretation (approximating sets of possible program states). It "executes" programs in an abstract domain—not on real hardware—to follow how variable values change. It allows to find sound upper bounds on loop iterations and identify infeasible paths that can never occur. SWEET operates on the ALF intermediate representation [25]; based on it, SWEET generates flow constraints that make WCET computation more automatic and accurate than manual annotation. However, SWEET's abstract interpretation can decrease precision, producing loose bounds that overestimate WCET.

Recognizing that neither purely algebraic nor purely abstract methods produce tight bounds, Knoop et al. [13] introduced r-TuBound, integrating pattern-based recurrence solving with Satisfiability Modulo Theory (SMT) solvers to calculate exact loop iteration counts. The approach: (i) converts multi-path loops into single-path equivalents using SMT, (ii) translates into recurrence equations, (iii) solves for closed forms using safe over-approximations, and (iv) uses SMT to calculate the smallest loop counter guaranteeing termination—precisely the trip count. r-TuBound achieves exactness only for restricted recurrence patterns and lacks multi-procedural analysis capabilities. When iteration variables are computed or modified across function calls, the tool cannot track data flow through call chains.

Modern WCET tools use multi-stage workflows: flow analysis (path and trip count calculation), low-level analysis (instruction timing modeling), and path analysis (final

WCET calculation by composition of the two). Frameworks like OTAWA [26] use the Implicit Path Enumeration Technique (IPET) [27] to create Integer Linear Programming (ILP) [28] constraints, derived from control flow, CPU pipeline, and instruction cache modelling data. Subsequently, it employs the ILP solver "lp_solve" [29], to resolve these constraints and determine the WCET.

Existing loop analysis techniques face fundamental limitations when applied to binary code. Critically, no existing approach addresses three binary-level challenges: (i) architecture-specific reasoning—current methods cannot leverage instruction-level semantics of architectures like AArch64; (ii) interprocedural data flow—tools cannot track induction variables through function call chains without manual annotations; and (iii) automation gaps—frameworks like OTAWA require manual loop bound annotations. These limitations create the need for automated, architecture-specific binary-level loop analysis capable of identifying natural loops, locating iteration variables despite compiler transformations, and handling multi-procedural scenarios.

# 4 Control Flow and Data Dependency Analysis

## 4.1 Disassembly and CFG Generation

The low-level nature of binary code makes it difficult to analyze program logic and execution paths in their raw form. Disassemblers solve this problem by transforming raw binary data into assembly code, providing a more interpretable representation of the program.

However, assembly code alone is not enough to analyze program structure. Detecting natural loops requires an explicit representation of control flow between program instructions. We use CFGs to capture this information, where nodes represent basic blocks and edges represent possible execution transfers caused by branch instructions. These representations provide the foundation for our subsequent analysis stages.

## 4.2 Loop Detection

Loop detection begins with constructing a dominator tree [12] from the CFG. A vertex $u$ is said to dominate a vertex $v$ if every path from the entry node of the CFG to $v$ passes through $u$. The dominator tree is a tree representation of this relation in which the parent of each vertex is the closest dominator of that vertex. While several algorithms exist for this computation, our method uses the Lengauer-Tarjan algorithm [30].

After identifying the dominators, we applied a similar process to find postdominators [12]. A vertex $u$ postdominates a vertex $v$ if every path from $v$ to an exit node passes through $u$. Analogously to dominators, the closest postdominator relationships form a postdominator tree. To compute postdominators, the algorithm first adds a virtual exit node with edges connecting all the original exit points to this new node, ensuring the modified graph has only one exit. It then reverses the direction of all edges in the CFG. This creates a modified graph with a single entry point, allowing us to apply the same dominator construction algorithm to detect postdominators.

After identifying dominators and postdominators, the algorithm traverses the dominator tree to detect back edges. During this traversal, it tracks the path from the root

to the current vertex, which corresponds to the set of dominators of that vertex. For each visited vertex, the algorithm checks whether any back edges originate from it in the original CFG—that is, edges that connect the vertex back to one of its dominators. Once all back edges have been collected in the CFG, the algorithm groups them by destination node. Each group represents a natural loop, which can have a single back edge or multiple back edges that lead to the same loop header.

The next step involves identifying the natural loop for each group of back edges. A natural loop consists of all vertices that can reach the back edge's source vertex without passing through its destination. The algorithm collects these vertices by traversing the transpose graph from the back edge's source while avoiding its destination vertex. This process is repeated for all back edges in the group to gather the complete set of loop vertices. The destination vertex of each back edge becomes the loop header, serving as the entry point for the loop.

Determining the nesting relationships between loops follows directly from the structure of natural loops, since any two natural loops are either nested or disjoint [12]. To determine whether one loop is nested within another, the algorithm checks whether the header of one loop belongs to the vertex set of the other loop. Because the header dominates all vertices of its loop, this condition is sufficient to establish loop containment. Based on these relationships, the algorithm constructs a loop nesting tree that represents the hierarchical organization of loops.

## 4.3 Data Dependency Analysis

Data dependency analysis begins with understanding how each instruction interacts with program data. Throughout this article, we use the term "variables" to refer to all storage locations in binary code that hold program data, such as registers, stack locations, and heap memory. Each instruction can read from or write to these storage locations. This read/write information is extracted from the disassembler, enabling subsequent data dependency analysis.

Computing reaching definitions determines which variable definitions can reach each program point (a position in the control flow where execution may pause, occurring immediately before or after each instruction). We implemented this using the worklist algorithm [31], which iteratively propagates definition information through the CFG until no further changes occur. Reaching definitions must be known for program points before and after each instruction. However, the number of instructions in a binary can be extremely large, making a direct instruction-level computation computationally expensive. To optimize this, the algorithm performs the analysis at the basic block level. Since basic blocks contain no internal control flow branches, the algorithm first computes the set of definitions generated by each basic block and calculates reaching definitions at the entry and exit of each basic block using the worklist algorithm. It then determines reaching definitions for individual instructions by taking the block's entry definitions and propagating them forward through each instruction in sequence.

Once reaching definitions are established, use-def chains can then be constructed directly. For each instruction, the algorithm intersects the set of reaching definitions at that program point with the set of variables the instruction reads, directly identifying

**Listing 1** Loop example in C.

```c
int i = 0;
while (i < 10) {
    printf("i = %d\n", i);
    i++;
}
```

**Listing 2** Corresponding AArch64 assembly code for the loop example.

```
    mov     w19, #0
0x698:
    mov     w2, w19
    add     w19, w19, #1
    bl      sym.imp.__printf_chk
    cmp     w19, #0xa
    b.ne    0x698
```

which definitions may supply values to each variable use. This approach builds the use-def relationship graph required for loop analyses.

# 5 Identification of Loop Iteration Variables

As mentioned above, the iteration variable is a counter that determines the number of loop iterations, updating its value in each iteration based on its previous value. We propose a three-step algorithm for identifying these variables within a loop:

1. Identify the conditional branch instructions that determine whether the loop continues or terminates.
2. Collect the instructions and variables that affect those conditional branch instructions.
3. Identify variables from this set that update their value based on their value from the previous iteration.

We illustrate the method using a simple source code loop example shown in Listing 1 and its corresponding assembly code in Listing 2.

## 5.1 Identification of Loop-Controlling Instructions

The first stage is to identify the instructions that control the loop's execution. For this purpose, we apply the algorithm described in [32], which detects iteration branches. An iteration branch in a loop is a conditional transfer of control where the choice between two outgoing transitions can directly or indirectly influence the number of loop iterations. Branches that directly affect the iteration count include those that either exit the loop, return to the loop header, or return to a block postdominated by the header. The algorithm produces a set of basic blocks. The last branch instruction

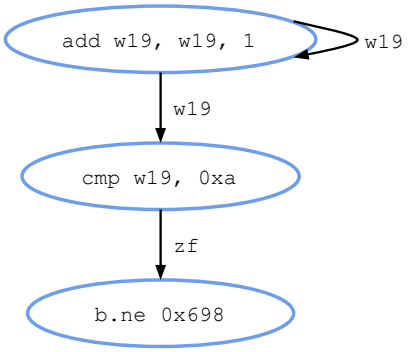



**Fig. 2** Use-def chains constructed for the branch instruction.

from each block is extracted to form the set of loop-controlling conditional branch instructions.

In the example shown in Listing 2, the branch instruction $b.ne\ 0x698$ determines whether the loop continues or terminates based on the flags set by the preceding *cmp* instruction. Since the loop contains only a single iteration branch, the resulting set of loop-controlling instructions consists of this instruction alone.

## 5.2 Collection of Instructions Affecting Conditional Branches

Once the conditional branch instructions are collected, the algorithm identifies the variables and instructions that influence their operation. For each branch instruction, it extracts the variables it reads and traces their use-def chains backward to their defining instructions. This process continues iteratively, following each definition to its predecessors, until the chain forms a cycle by returning to an already visited instruction. The algorithm collects these instruction chains for all variables associated with the loop's conditional branch instructions.

Figure 2 shows the use-def chains for the conditional branch instruction from Listing 2. Following these chains backward, we observe that the instruction $add\ w19, w19, \#1$ forms a self-cycle in the use-def chain, where it both uses and defines the same variable.

## 5.3 Identification of Self-Updating Variables

In the final stage, the algorithm analyzes each chain collected in the previous step to identify variables that meet the criteria of iteration variables. Specifically, it examines the cyclic parts of these chains to determine whether they contain instructions that cause a variable to update its value based on its previous value. Variables modified by such instructions satisfy both conditions required for iteration variables: they influence whether the loop continues or terminates, and they update self-dependently at each iteration. These variables are reported as the iteration variables of the loop.

In the example shown in Figure 2, the instruction $add\ w19, w19, \#1$ increments register $w19$ based on its previous value, forming a cycle in the use-def chain.

Since this register influences the loop-controlling conditional branch and updates self-dependently, $w19$ is identified as the iteration variable of the loop.

# 6 Identification of Loop Iteration Counts

The next phase of our analysis is dedicated to calculating the precise number of loop iterations. This analysis is restricted to loops that satisfy a specific structural model described below. We present a three-step algorithm for identifying loop iteration counts based on previously identified iteration variables:

1. Verify that the loop satisfies the structural requirements for analysis.
2. Extract the iteration variable's initialization, update operation, and comparison condition.
3. Construct and solve the loop equation to determine the iteration count.

## 6.1 Verification of Loop Structure

At this stage, the algorithm verifies that the loops match the required structural pattern supported by our approach. The loops selected for analysis must meet the following conditions:

- Single iteration variable: the loop must have only one iteration variable that controls its progress and termination.
- Single loop condition: the loop must have a single conditional branch that controls its continuation or termination, without multiple comparisons or combined conditions.
- Linear use and update of the iteration variable: the iteration variable must appear linearly both in the loop condition and in its update expression.

For clarity, the supported loop structure can be represented as loop shown in Equation (1). This is a conceptual representation at the source-code level; in binary code, the structure is more complex.

$$for\ (int\ i = a;\ b_1 + b_2 * i\ [comparison\ sign]\ b_3;\ i = c * i + d) \tag{1}$$

where:

- $i$ — iteration variable,
- $a$ — initial value of the iteration variable,
- $b_1, b_2, b_3$ — constants defining the loop termination condition,
- $[comparison\ sign]$ — one of $==$, $!=$, $>$, $>=$, $<$, or $<=$,
- $c, d$ — constants defining the update operation.

At this stage, the algorithm verifies each of these requirements and filters out loops that do not satisfy them. Additionally, it performs supplementary checks, such as ensuring that all instructions involving the iteration variable execute unconditionally in every iteration and are not part of any conditional branches. In the following sections, this loop form is used as the basis for iteration count analysis.

## 6.2 Extraction of Iteration Variable Parameters

The next step is to extract the iteration variable's initial value, update operation, and comparison parameters. In our representation (Equation (1)), this corresponds to determining the values of $a$, $b_1$, $b_2$, $b_3$, [*comparison sign*], $c$, and $d$. If any parameter cannot be determined, the loop is excluded from analysis. The following subsections describe the techniques used to extract these parameters.

### 6.2.1 Emulation of Instructions

To determine variable values at specific program points, we use instruction-level emulation. During emulation, the behavior of each instruction is simulated, updating register and memory states according to their effects. This allows us to reproduce how instructions modify the program state and to obtain the required variable values for further analysis.

### 6.2.2 Determination of Iteration Variable's Initial Value

The initial value of the iteration variable is its value upon loop entry. This value may be computed through a sequence of operations, may depend on other variables, or may be passed as a function argument. To determine the initial value, the algorithm traverses the use-def chain backward from the loop entry, collecting all instructions that define the iteration variable before the loop entry. The traversal continues until no further definitions are found. If the final instruction of the chain is a constant assignment, the collected instructions are emulated to compute the initial value. However, if multiple definition paths are encountered or the final definition is non-constant, the analysis terminates, as the value cannot be uniquely determined. If the chain reaches a function argument, interprocedural analysis is performed starting from the function call site.

### 6.2.3 Determination of Comparison Parameters

In this stage, the algorithm identifies the comparison operator and the comparison parameters $b_1$, $b_2$, and $b_3$ used in the loop condition. To find them, the algorithm traces backward through the use-def chain of instructions that starts from the comparison instruction. The comparison sign is extracted directly from the branch instruction type (e.g., $b.ne$ yields $!=$, $b.lt$ yields $<$). If operands are reversed, the comparison operator is logically flipped. Parameter $b_3$ is obtained by backward emulation starting from the comparison instruction, using the same approach as initial value determination. The algorithm also analyzes the instruction chain that executes after the iteration variable update and before the comparison. This chain may contain additional computations applied to the iteration variable, which are represented in the formula as $b_1$ and $b_2$. The coefficients $b_1$ and $b_2$ are determined using the recurrence analysis method described in Subsection 6.2.4.

### 6.2.4 Determination of Iteration Variable's Transformation Parameters

The iteration variable updates according to the equation $i_n = c \cdot i_{n-1} + d$, where $i_n$ is the value of the iteration variable after the $n$-th iteration. To determine the constants $c$ and $d$, the algorithm solves the following system of equations:

$$\begin{cases} i_1 = c \cdot i_0 + d \\ i_2 = c \cdot i_1 + d \end{cases} \tag{2}$$

where $i_0$ is the initial value of the iteration variable, $i_1$ and $i_2$ are determined using a similar approach to finding the initial value of the iteration variable, but this time the algorithm emulates instructions that change the iteration variable inside the loop.

$$\begin{cases} c = \frac{i_2 - i_1}{i_1 - i_0} \\ d = i_1 - c \cdot i_0 \end{cases} \tag{3}$$

The same approach is used to determine the constants in the expression that define how the iteration variable changes before reaching the comparison instruction. As a result, we obtain the constants $c$ and $d$ describing the transformation of the iteration variable.

## 6.3 Construction and Solution of the Loop Equation

Using calculated parameters, we construct an equation that models the loop's iteration behavior and solve it to obtain the iteration count. The loop behavior is modeled as a recurrence relation, where $i_n$ represents the value of the iteration variable after $n$ iterations:

$$\begin{cases} i_{n+1} = c \cdot i_n + d \\ i_0 = a \end{cases} \tag{4}$$

The closed-form solution for $i_n$ has two cases:

- **Case 1:** $c \neq 1$

$$i_n = \alpha \cdot c^n + \beta, \tag{5}$$

where $\alpha$ and $\beta$ satisfy:

$$\begin{cases} \alpha + \beta = a \\ \alpha \cdot c + \beta = a \cdot c + d \end{cases} \tag{6}$$

- **Case 2:** $c = 1$

$$i_n = d \cdot n + a, \tag{7}$$

This recurrence-based formulation of iteration behavior follows the idea introduced in r-TuBound [13], adapted here for binary analysis.

To find the iteration count, we determine the value of $n$ for which the loop condition is satisfied at $i_n$ but fails at $i_{n+1}$. To determine this, we solve the loop termination equation for $n$:

$$b_1 + b_2 \cdot i_n = b_3 \tag{8}$$

Substituting the closed-form expression for $i_n$ yields a real-valued solution $n_{\text{real}}$. The actual iteration count is then determined by rounding $n_{\text{real}}$ to a nearby integer, with the specific rounding direction depending on the comparison operator used in the loop condition.

During this process, the algorithm also checks for infinite loops (loops where execution never exits the loop body once entered). When the loop condition is satisfied by the initial value of the iteration variable, the algorithm verifies whether the loop will terminate by checking the following conditions:

- **Case 1: $x \notin \mathbb{Z}$ and the comparison operator is "$\neq$".** The equality cannot be satisfied for any integer $n$, so the loop condition never becomes false.
- **Case 2: $i_n$ is constant.** Once the loop condition is satisfied, it remains true for all subsequent iterations.
- **Case 3: $i_n$ is increasing, and the comparison operator is "$>$" or "$\geq$".** The iteration variable moves away from the boundary value.
- **Case 4: $i_n$ is decreasing, and the comparison operator is "$<$" or "$\leq$".** The iteration variable moves away from the boundary value.

Thus, the exact number of loop iterations is obtained by solving the constructed equation and checking for infinite loop conditions. This completes the iteration count analysis for the loop.

# 7 Implementation

The tool was implemented as a C library and tested on AArch64 architecture binaries. Assembly code, CFGs, and other binary artifacts are extracted using the Radare2 [33] disassembler (version 5.9.0) through its C API.

For instruction emulation during the iteration count identification stage, we utilize Radare2's ESIL (Evaluable Strings Intermediate Language) emulation engine. ESIL provides architecture-independent partial emulation by translating assembly instructions into an intermediate representation that can be evaluated across all supported platforms. This approach enables us to compute constant values through sequences of operations without requiring full system emulation, significantly reducing computational overhead.

The reliability of the implementation was validated through comprehensive testing using tools for memory error detection and leak identification [34–37]. All experiments were conducted on a system equipped with an 8-core Intel Core i7-11700 CPU (2.50 GHz) and 16 GB of RAM, running Ubuntu 22.04 LTS.

# 8 Results

This section presents the results of applying our implemented algorithms to diverse binary programs. We separately analyze the performance of loop detection and iteration count identification, focusing on two main aspects: accuracy and efficiency.

| Project | Binary Size | Functions | Overall Time | Analysis Time |
|---|---|---|---|---|
| Rsync | 2.3 MB | 900 | 44 s | 3.1 s |
| Git | 23.0 MB | 6,366 | 341 s | 11.7 s |
| NeoVim | 26.1 MB | 11,741 | 482 s | 15.8 s |

**Table 1** Runtime performance on large open-source projects.

Accuracy is measured using precision and recall metrics, while efficiency is evaluated by analyzing runtime distribution across different analysis stages. Establishing ground truth for loop iteration counts in large real-world applications is particularly challenging. For projects such as Rsync, Git, and NeoVim, which contain thousands of functions and complex interactions between modules, manually determining the exact iteration count of every loop is practically infeasible. Consequently, these projects are used primarily to evaluate the scalability and performance of our analysis rather than its absolute accuracy with respect to iteration counts. To evaluate the correctness of iteration count identification, we rely on the Mälardalen WCET benchmark suite, for which we manually established the ground truth for all analyzed loops. The relatively small size and well-structured nature of these benchmarks make such manual validation feasible, in contrast to large industrial-scale applications.

Our evaluation includes comparative analysis with established tools Angr (version 9.2.201) [38] and LLVM (release 18.1.8) [20]. Results are presented for three large open-source projects compiled with default compilation settings for the AArch64 architecture: Rsync [39], Git [40], and NeoVim [41]. For iteration count identification, we additionally used the Mälardalen WCET benchmark suite [42], which provides standardized test cases for loop analysis. Detailed results and analysis for each algorithm are presented in the following subsections.

## 8.1 Performance Analysis

In this section, we evaluate the runtime performance of our tool's iteration count detection, which includes loop detection, iteration variable identification, and the actual iteration count analysis. The experimental results indicate that disassembly and CFG construction performed by Radare2 dominate the total execution time. In contrast, the subsequent analysis phases—loop detection, reaching definitions computation, use-def chains construction, and iteration count detection—require minimal computational overhead.

Table 1 presents the performance results for three large open-source projects. The "Overall Time" column represents the complete end-to-end execution time, including Radare2's preprocessing, while the "Analysis Time" column isolates the runtime of our core algorithms alone.

## 8.2 Natural Loop Detection

We compared results of our loop detection algorithm against Angr, an established binary analysis framework. A key architectural difference between the two tools lies in CFG construction: Radare2 constructs separate CFGs for each function, while Angr

| Project | Our Tool (Loops) | Angr (Loops) | Common Loops |
|---|---|---|---|
| Rsync | 681 | 663 | 646 |
| Git | 5210 | 4810 | 4775 |
| NeoVim | 6145 | 6244 | 6089 |

**Table 2** Natural loop detection comparison with Angr.

| Project | Natural Loops | Top-level Natural Loops | Top-level Cycles |
|---|---|---|---|
| Rsync | 681 | 441 | 492 |
| Git | 5210 | 3291 | 3617 |
| NeoVim | 6145 | 4642 | 4724 |

**Table 3** Statistics on natural loops and cycles.

builds a unified CFG for the entire binary. This fundamental difference affects control flow analysis, leading to variations in loop detection results.

Despite these architectural differences, the loop detection results show strong agreement between both tools. Table 2 presents the comparison across three test programs, showing the total number of loops detected by each tool and the number of loops identified by both.

### 8.2.1 Comparison of Cycles and Natural Loops

Cycles are a generalization of natural loops. While every natural loop forms a cycle in the CFG, not every cycle corresponds to a natural loop. The key difference is that natural loops have a single entry point, whereas cycles may have multiple entries.

In some cases, loops in the original program are transformed into cycles of this kind, primarily due to compiler optimizations. Without the single-entry guarantee of natural loops, analyzing cycles becomes significantly more complex in subsequent analysis stages, as tools cannot establish consistent entry conditions when entering a cycle.

This design choice leads to gaps in coverage for certain structures in binary code. To understand how this limitation affects our analysis, we measured the percentage of top-level cycles (cycles not contained within other cycles) that are not natural loops. We implemented an algorithm for detecting top-level cycles in CFGs. We applied this algorithm, along with our natural loop detection method, to a diverse set of real-world projects compiled with varying optimization levels. Our results (Table 3) show that roughly 90% of top-level cycles across evaluated binaries are natural loops, indicating that our tool analyzes a significant portion of cyclic code structures. This finding aligns with the fact that structured programming languages generally produce loops with a single entry point, while compiler optimizations can introduce control flow patterns with multiple entries.

| Tool | True Positive | False Positive | False Negative | Precision | Recall |
|---|---|---|---|---|---|
| Our tool | 103 | 0 | 68 | 100% | 60.2% |
| RetDec + LLVM | 76 | 0 | 95 | 100% | 44.4% |

**Table 4** Iteration count detection comparison with RetDec + LLVM on Mälardalen WCET benchmark.

### 8.3 Identification of Loop Iteration Counts

The algorithm for identifying iteration counts was tested on the Mälardalen WCET benchmark suite [42], which comprises 35 benchmark programs containing a total of 171 source code loops. The binaries were compiled with -O0 optimization for AArch64. Compiling without optimizations preserves a direct mapping to source-level semantics, facilitating ground-truth validation between the binary-level analysis and the source code.

We ran our tool on all 35 program binaries and manually verified the results for correctness. Our tool detected 170 of the 171 source code loops. The single undetected case was not a natural loop because a jump from outside the loop into the loop body violated the single-entry property of natural loops. Our tool successfully determined iteration counts for 103 of the 170 detected loops. All detected iteration counts were manually confirmed to be correct, yielding 100% precision on these test cases. Cases where iteration counts were not detected include loops with multiple exit conditions, loops with complex floating-point operations in the exit condition, and cases where inner loop iteration counts depend on the iteration variable of the outer loop.

We also compared our tool's results with LLVM's trip count analysis. To execute LLVM passes on binary code, we first converted the binaries into LLVM bytecode using RetDec (release v5.0) [43], applied mem2reg optimizations to the generated bytecode, and then executed LLVM's trip count analysis pass. LLVM successfully determined correct trip counts for 76 out of the 171 loops. In comparison, our tool correctly determined all 76 loops identified by LLVM and successfully analyzed an additional 27 loops, thereby achieving broader coverage while maintaining 100% precision. The complete evaluation results on the Mälardalen WCET benchmark suite [42] are summarized in Table 4.

Additionally, we compared our tool and LLVM on the real-world projects mentioned above: Rsync, Git, and NeoVim. However, RetDec failed to decompile Git and NeoVim, limiting the comparison to Rsync only. On the Rsync project, our tool detected iteration counts for 12 loops, and LLVM detected 14; of these, 4 loops were detected by both tools. This means our tool uniquely identified 8 iteration counts that LLVM missed. Our tool's inability to detect the 10 cases that LLVM successfully identified was primarily due to currently unsupported binary instructions rather than limitations of the underlying algorithm. These results are presented in Table 5.

## 9 Conclusion and Future Work

This paper introduced an automated interprocedural static analysis framework for loop detection, iteration variable identification, and iteration count determination in binary

| Project | Our Tool (IC) | RetDec + LLVM (IC) | Common IC |
|---|---|---|---|
| Rsync | 12 | 14 | 4 |
| Git | 68 | X | X |
| NeoVim | 192 | X | X |

**Table 5** Iteration count detection comparison with RetDec + LLVM on open-source projects.

code. Our algorithms have been tested on a diverse set of binaries, achieving 100% precision and 60.2% recall on the Mälardalen WCET benchmark suite and successfully tracking 27 more loops than the RetDec+LLVM baseline. The results show that our approach successfully handles many commonly occurring loop structures in real-world binaries.

The developed methods have significant implications for code analysis and optimization tools, such as those used in WCET analysis. The results of our research can be used to improve the efficiency of such tools.

Several directions for future work remain. First, the method should be extended to support additional processor architectures beyond AArch64. Second, broadening the types of analyzable loop structures—including loops with multiple exits, nested conditions, and irregular control flow—would increase the method's practical applicability.

**Author Contributions.** All authors contributed to the design and implementation of the study, analyzed and interpreted the results, reviewed and approved the final version of the manuscript, and agree to be accountable for all aspects of the work. H.A. supervised the research, drafted Sections 1, 2, reviewed and revised the manuscript. G.K. drafted Sections 4 and 7. S.H. drafted Sections 5 and 6. H.H. drafted Sections 3, 9.

**Funding.** The research was supported by the Higher Education and Science Committee of Ministry of Education, Science, Culture and Sport of the Republic of Armenia (Research project № 25RG-1B142).

**Data Availability.** The Mälardalen WCET benchmark suite analysed during the current study is publicly available at https://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

## Declarations

**Conflict of Interest.** The authors declare no competing interests.

**Ethics Approval.** Not applicable.

## References

[1] Knuth, D.E.: An empirical study of fortran programs. Software: Practice and experience **1**(2), 105–133 (1971)

[2] Allen, F.E., Cocke, J.: A catalogue of optimizing transformations. In: Rustin, R. (ed.) Design and Optimization of Compilers, pp. 1–30. Prentice-Hall, Englewood Cliffs, NJ, USA (1972)

[3] Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. **26**(4), 345–420 (1994) https://doi.org/10.1145/197405.197406

[4] Ren, X., Ho, M., Ming, J., Lei, Y., Li, L.: Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021, pp. 142–157. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454035

[5] Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ISSTA '09, pp. 225–236. Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1572272.1572299

[6] Rawat, S., Mounier, L.: Finding buffer overflow inducing loops in binary executables. In: Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability. SERE '12, pp. 177–186. IEEE Computer Society, USA (2012). https://doi.org/10.1109/SERE.2012.30

[7] Wei, T., Mao, J., Zou, W., Chen, Y.: A new algorithm for identifying loops in decompilation. In: Proceedings of the 14th International Conference on Static Analysis. SAS'07, pp. 170–183. Springer, Berlin, Heidelberg (2007)

[8] Collins, E.D., Miller, B.P.: A loop-aware search strategy for automated performance analysis. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J. (eds.) High Performance Computing and Communications, pp. 573–584. Springer, Berlin, Heidelberg (2005)

[9] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3) (2008) https://doi.org/10.1145/1347375.1347389

[10] Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., Lisper, B.: Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis. In: Rochange, C. (ed.) 7th International Workshop on Worst-Case Execution Time Analysis (WCET'07). Open Access Series in Informatics (OASIcs), vol. 6, pp. 1–6. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2007). https://doi.org/10.4230/OASIcs.WCET.2007.1194 .

https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2007.1194

[11] Machiry, A., Redini, N., Gustafson, E., Fratantonio, Y., Choe, Y.R., Kruegel, C., Vigna, G.: Using loops for malware classification resilient to feature-unaware perturbations. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACSAC '18, pp. 112–123. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274731

[12] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)

[13] Knoop, J., Kovács, L., Zwirchmayr, J.: Symbolic loop bound computation for wcet analysis. In: Clarke, E., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics, pp. 227–242. Springer, Berlin, Heidelberg (2012)

[14] Burnim, J., Jalbert, N., Stergiou, C., Sen, K.: Looper: Lightweight detection of infinite loops at runtime. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering. ASE '09, pp. 161–169. IEEE Computer Society, USA (2009). https://doi.org/10.1109/ASE.2009.87

[15] Carbin, M., Misailovic, S., Kling, M., Rinard, M.C.: Detecting and escaping infinite loops with jolt. In: Proceedings of the 25th European Conference on Object-Oriented Programming. ECOOP'11, pp. 609–633. Springer, Berlin, Heidelberg (2011)

[16] Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In: 2006 27th IEEE International Real-Time Systems Symposium (RTSS'06), pp. 57–66 (2006). https://doi.org/10.1109/RTSS.2006.12

[17] Wägemann, P., Distler, T., Hönig, T., Janker, H., Kapitza, R., Schröder-Preikschat, W.: Worst-case energy consumption analysis for energy-constrained embedded systems. In: 2015 27th Euromicro Conference on Real-Time Systems, pp. 105–114 (2015). https://doi.org/10.1109/ECRTS.2015.17

[18] Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), pp. 81–90 (2006). https://doi.org/10.1109/RTAS.2006.17

[19] Wolfe, M.: Beyond induction variables. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. PLDI '92, pp. 162–174. Association for Computing Machinery, New York, NY, USA (1992). https://doi.org/10.1145/143095.143131

[20] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program

analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86 (2004). IEEE

[21] Free Software Foundation: Using the GNU Compiler Collection. Free Software Foundation, (2025). Free Software Foundation. Version 15.2. https://gcc.gnu.org/onlinedocs/gcc/

[22] Bachmann, O., Wang, P.S., Zima, E.V.: Chains of recurrences—a method to expedite the evaluation of closed-form functions. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation. ISSAC '94, pp. 242–249. Association for Computing Machinery, New York, NY, USA (1994). https://doi.org/10.1145/190347.190423

[23] Engelen, R.v.: Efficient symbolic analysis for optimizing compilers. In: Proceedings of the 10th International Conference on Compiler Construction. CC '01, pp. 118–132. Springer, Berlin, Heidelberg (2001)

[24] Lisper, B.: Sweet – a tool for wcet flow analysis (extended abstract). In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, pp. 482–485. Springer, Berlin, Heidelberg (2014)

[25] Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., Källberg, L.: ALF - A Language for WCET Flow Analysis. In: Holsti, N. (ed.) 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09). Open Access Series in Informatics (OASIcs), vol. 10, pp. 1–11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2009). https://doi.org/10.4230/OASIcs.WCET.2009.2279 . https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2009.2279

[26] Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An open toolbox for adaptive WCET analysis. In: Min, S.L., Pettit, R., Puschner, P., Ungerer, T. (eds.) Software Technologies for Embedded and Ubiquitous Systems, pp. 35–46. Springer, Berlin, Heidelberg (2010)

[27] Li, Y.-T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. SIGPLAN Not. **30**(11), 88–98 (1995) https://doi.org/10.1145/216633.216666

[28] Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Chichester, UK (1986)

[29] Berkelaar, M., Eikland, K., Notebaert, P.: lp_solve: Open source (Mixed-Integer) Linear Programming system. http://lpsolve.sourceforge.net/. Version 5.5 (2004)

[30] Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. **1**(1), 121–141 (1979) https://doi.org/10.1145/357062.357071

[31] Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 194–206 (1973). ACM

[32] Healy, C., Sjodin, M., Rustagi, V., Whalley, D.: Bounding loop iterations for timing analysis. In: Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245), pp. 12–21 (1998). https://doi.org/10.1109/RTTAS.1998.683183

[33] Radare2 Development Team: Radare2: Unix-Like Reverse Engineering Framework and Command-Line Toolset. https://rada.re/n/radare2.html (2006)

[34] Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007) https://doi.org/10.1145/1273442.1250746

[35] Aslanyan, H., Movsisyan, H., Hovhannisyan, H., Gevorgyan, Z., Mkoyan, R., Avetisyan, A., Sargsyan, S.: Combining static analysis with directed symbolic execution for scalable and accurate memory leak detection. IEEE Access **12**, 80128–80137 (2024) https://doi.org/10.1109/ACCESS.2024.3409838

[36] Belevantsev, A.: Multilevel static analysis for improving program quality. Programming and Computer Software **43**, 321–336 (2017) https://doi.org/10.1134/S0361768817060044

[37] Sargsyan, S., Hakobyan, J., Mehrabyan, M., Mkoyan, R., Sahakyan, V., Melkonyan, V., Arutunian, M., Fahradyan, A., Avetisyan, A.: Advanced grammar-based fuzzing. In: 2022 Ivannikov Memorial Workshop (IVMEM), pp. 61–64 (2022). https://doi.org/10.1109/IVMEM57067.2022.9983953

[38] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 138–157 (2016). https://doi.org/10.1109/SP.2016.17

[39] Tridgell, A., Mackerras, P.: Rsync: A Fast, Versatile, Remote (and Local) File-Copying Tool. https://rsync.samba.org/ (1996)

[40] Torvalds, L., Hamano, J.C.: Git: A Distributed Version Control System. https://git-scm.com/ (2005)

[41] Neovim Development Team: Neovim: Hyperextensible Vim-Based Text Editor.

https://neovim.io/ (2014)

[42] Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET Benchmarks: Past, Present And Future. In: Lisper, B. (ed.) 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010). Open Access Series in Informatics (OASIcs), vol. 15, pp. 136–146. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2010). https://doi.org/10.4230/OASIcs.WCET.2010.136 . https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2010.136

[43] Avast Software: RetDec: A Retargetable Machine-Code Decompiler. https://retdec.com/ (2017)