Sound Termination and Non-termination Analysis of C Programs with Bit-Precise Bounded Semantics and Advanced Constructs

NEGAR FATHI, University of Nebraska-Lincoln, USA
HIROSHI UNNO, Tohoku University, Japan
TACHIO TERAUCHI, Waseda University, Japan
RAHUL PURANDARE, University of Nebraska-Lincoln, USA

Program termination and non-termination analysis is a foundational problem in formal verification with important implications for software safety and reliability. Despite extensive research, existing techniques struggle with real-world C programs that manipulate complex data types such as pointers, arrays, and structures, or that perform low-level operations such as bitwise arithmetic and bounded integer computations. This paper introduces Athena, a framework for sound termination and non-termination analysis of C programs that models finite-width and bit-precise integer semantics and supports advanced constructs. Athena combines pointer-to-array rewriting, bounded integer semantics enforced via modulo arithmetic or bit-vector semantics, and an extended translation to Labeled Transition Systems (LTS), yielding structured, analyzable representations suitable for logic-based reasoning. Our analysis engine builds on MuVal, a modular verification engine based on the first-order fixpoint logic μCLP with background theories, and extends it with support for array, tuple, and bit-vector theories in ranking function synthesis and recurrent set detection. We evaluate Athena on the 2024 Termination Competition (TermCOMP) and on 117 real-world benchmarks featuring 445 non-termination bugs, after excluding benchmarks that rely on undefined behavior. It achieves 60.95% correctness on the real-world benchmarks and 76.28% on TermCOMP, while producing zero wrong results across both suites. These results highlight Athena's strong combination of precision and soundness for the termination and non-termination analysis of complex C programs.

CCS Concepts: • **Software and its engineering → Formal software verification**.

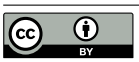
Additional Key Words and Phrases: Program Termination and Non-termination, Ranking Functions, Recurrent Sets, Finite-Width Semantics, Program Rewriting.

ACM Reference Format:
Negar Fathi, Hiroshi Unno, Tachio Terauchi, and Rahul Purandare. 2026. Sound Termination and Non-termination Analysis of C Programs with Bit-Precise Bounded Semantics and Advanced Constructs. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE198 (July 2026), 24 pages. https://doi.org/10.1145/3808205

1 Introduction

Program termination and non-termination analysis plays a critical role in computer science, particularly in software engineering and formal methods. The goal of this analysis is to determine whether a program will eventually halt or may continue executing indefinitely under certain conditions [19]. Software that fails to terminate as expected can cause system crashes, exhaust hardware resources without making progress, and often requires forceful termination [60]. However, distinguishing

Authors' Contact Information: Negar Fathi, University of Nebraska-Lincoln, Lincoln, USA, nfathi2@huskers.unl.edu; Hiroshi Unno, Tohoku University, Sendai, Japan, hiroshi.unno@acm.org; Tachio Terauchi, Waseda University, Tokyo, Japan, terauchi@waseda.jp; Rahul Purandare, University of Nebraska-Lincoln, Lincoln, USA, rahul@unl.edu.



ACM 2994-970X/2026/7-ARTFSE198
https://doi.org/10.1145/3808205

between a program that is trapped in an unproductive loop and one that is performing legitimate, complex computations at runtime remains extremely challenging.

Extensive research has been conducted on the termination [1, 4, 7–9, 16, 17, 19, 21, 27, 37, 40, 46] and non-termination [2, 13, 18, 26, 27, 35, 39, 40, 58] analysis of imperative programs. These studies primarily focus on synthesizing ranking functions [5, 7, 41, 45, 46, 57], which prove program termination, or identifying recurrent sets [3, 13, 18, 26, 39], which characterize potentially infinite program executions. However, existing approaches exhibit significant limitations when applied to real-world software. They struggle with complex data types such as pointers, arrays, and structures, as well as low-level operations like bitwise arithmetic and bounded integer computations. Most were designed for small or synthetic benchmarks and lack the scalability needed for industrial-scale codebases [20, 49]. Moreover, although many tools offer formal soundness guarantees within the semantics they are designed to support, the abstractions they rely on often omit critical features of real-world C programs. As a result, these tools may produce incorrect termination or non-termination results when applied to practical code, due to a mismatch between their abstract models and real-world C behavior [49].

These limitations are further evidenced by a large-scale empirical study conducted by Shi et al. [24, 49], which examined non-termination bugs in real-world C/C++ software. By analyzing 3,142 commits across 1,600 GitHub projects, they identified 445 genuine bugs involving pointers, arrays, bitwise operations, and overflow/underflow behaviors involving finite-width arithmetic effects. Their evaluation of leading tools—AProVE [25], UAutomizer [32], CPAchecker [6], 2LS [48], and T2 [10]—revealed that most failed to detect over half of these bugs, with average detection rates below 50%. These findings highlight a critical gap between current techniques and the demands of real-world software, underscoring the need for more expressive analyses that model realistic C program behavior.

To address these challenges, we propose Athena, a framework built on MuVal [38, 55], a modular verification engine for first-order fixpoint logic formulas in μCLP with background theories. MuVal supports both inductive and coinductive reasoning via a primal-dual fixpoint strategy, enabling modular analysis of properties such as termination and non-termination. However, it lacks support for critical C features such as pointers, arrays, structures, and bitwise operations. We extend MuVal with ranking function synthesis and recurrent set detection over array, tuple, and bit-vector theories. On the front end, we introduce a transformation that rewrites pointer operations into array indexing to enable structured and analyzable memory access, and provide two complementary modes for bounded integer semantics to ensure soundness: *modulo arithmetic*, which augments programs with explicit wraparound operations to model overflow and underflow under finite-width integer semantics, and *bit-vector semantics*, which extract type information to annotate the Labeled Transition System (LTS) for bit-precise reasoning. Additionally, we enhance llvm2KITTeL [22] with new abstractions for arrays, structures, and bitwise operations, enabling the generation of LTSs from the LLVM Intermediate Representation (LLVM IR) [54] of the input program. Athena is evaluated on both standard benchmarks, such as TermCOMP [52], and the real-world benchmark suite constructed by Shi et al. [24, 49], after excluding benchmarks whose expected behavior relies on undefined behavior, demonstrating that it effectively addresses complex non-termination challenges in real-world C programs.

In this work, we make the following key contributions:

(1) We propose Athena, the first framework for termination and non-termination analysis of C programs that models finite-width and bit-precise integer semantics and supports advanced constructs, including pointers, arrays, structures, and bitwise operations, by unifying source-level pointer-to-array rewriting, bounded integer semantics, and μCLP-based reasoning in a single end-to-end pipeline.

```
1  extern int __VERIFIER_nondet_int(void);
2  int flag = 0;
3  int fopen_or_warn() {
4    flag++;
5    ...
8    else return i;
9  }
10 int main() {
11   int len = __VERIFIER_nondet_int();
12   ...
18   goto GOT_NEW_FILE;
19   do {
20     if(*argv > 0) {
21       fp = fopen_or_warn();
22       if(fp == 0) continue;
23     }
24     argv++;
25 GOT_NEW_FILE:
26     fp++;
27   } while(*argv);
28   return 0;
29 }
```

(a) Motivating example 1: `Missing_Iterator_Update_3_NT`.

```
1  extern unsigned short
       __VERIFIER_nondet_ushort(void);
2  int main() {
3    unsigned short int s, seqnum, len;
4    seqnum = __VERIFIER_nondet_ushort();
5    len = __VERIFIER_nondet_ushort();
6    for(s = seqnum; s < seqnum + len; s++)
7      ;
8    return 0;
9  }
```

(b) Motivating example 2: `Type_Conversion_in_Comparison_1_NT`.

Fig. 1. Motivating examples adapted from Shi et al.'s benchmark suite [24, 49].

(2) We extend MuVal by incorporating support for array, tuple, and bit-vector theories into its ranking function synthesis and recurrent set detection, enabling precise reasoning about low-level and complex memory operations.
(3) We present a pointer-to-array rewriting technique that transforms pointer-based memory access into structured array indexing to facilitate analysis. The transformation is a reusable front-end: it emits standard C within our target fragment, enabling reuse as a preprocessing step by analyses whose front-ends support this fragment.
(4) We support bounded integer semantics through two alternative modes to ensure soundness: *modulo arithmetic*, which introduces explicit wraparound operations, and *bit-vector semantics*, which enable bit-precise reasoning via type-based LTS annotations.
(5) We enhance llvm2KITTeL with new abstractions for arrays, structures, and bit-level operations, enabling translation into analyzable LTS representations.
(6) We demonstrate through extensive evaluation that Athena achieves superior correctness on real-world benchmarks, competitive performance on TermCOMP, and uniquely produces zero wrong results.

The remainder of this paper is organized as follows. Section 2 presents motivating examples that illustrate the challenges of analyzing C programs with realistic behavior and advanced constructs. Section 3 reviews related work. Section 4 introduces our approach, and Section 5 details the implementation of Athena. Section 6 reports our experimental results. Finally, Section 7 concludes and outlines directions for future work.

2 Motivating Examples

Shi et al. [24, 49] introduced a benchmark suite of simplified C/C++ programs derived from 445 real-world non-termination bugs, identified through a large-scale analysis of 3,142 commits across 1,600 GitHub repositories. From this suite, we select two representative programs that do not rely on undefined behavior to illustrate the limitations of existing tools and the strengths of Athena: `Missing_Iterator_Update_3_NT` (Figure 1a) and `Type_Conversion_in_Comparison_1_NT` (Figure 1b).

The program in Figure 1a initializes an array of nondeterministic length, traverses it using pointer arithmetic, and repeatedly calls `fopen_or_warn` inside a loop controlled by an explicit `goto`. While

the program is non-terminating, it incorporates advanced features—array allocation with nondeterministic size, pointer arithmetic and aliasing, `goto`-based control flow, and function calls with side effects—that challenge state-of-the-art tools: AProVE and 2LS return `unknown`, indicating that they cannot prove either termination or non-termination; UAutomizer times out after 20 minutes; and CPAchecker crashes due to a bit-width mismatch in alias analysis. In contrast, Athena proves non-termination by rewriting pointer-based memory accesses into array-based form (Section 4.2), eliminating aliasing while preserving the program's memory behavior. The transformed program is then translated into a Labeled Transition System (LTS) using our extended version of llvm2KITTeL, which supports arrays, structures, and other low-level constructs (Section 4.4). This enables Athena to synthesize recurrent sets and soundly prove non-termination.

The program in Figure 1b becomes non-terminating when `seqnum + len` exceeds the maximum representable value of `unsigned short`. Since `seqnum + len` is computed after integer promotions whereas `s` remains finite-width and wraps around, the guard `s < seqnum + len` never becomes false. Such low-level arithmetic behavior is common in real-world C programs, where finite-width wraparound and boundary effects can significantly affect control flow. However, existing tools struggle to account for this interaction between finite-width arithmetic and integer promotions: Proton [43, 44] and CPAchecker incorrectly conclude termination, while AProVE, UAutomizer, and 2LS are inconclusive. Athena addresses this limitation through two user-selectable modes: (1) *modulo arithmetic* (Section 4.3.1), which encodes bounded integer semantics by instrumenting arithmetic operations to model wraparound under finite-width integer semantics; and (2) *bit-vector semantics* (Section 4.3.2), where a type information extraction pass records variable widths and operator signedness and attaches these annotations to the LTS during its construction, enabling bit-precise reasoning. Both modes ensure sound reasoning about finite-width arithmetic and allow Athena to prove non-termination in the motivating example by identifying recurrent sets.

3 Related Work

While program termination and non-termination have been extensively studied, few tools provide sound and scalable analysis for real-world C programs involving pointers, arrays, structures, bit-level operations, and bounded integers. Many methods assume unbounded arithmetic, simplify memory behavior, or target restricted C subsets, limiting their applicability—especially to embedded and systems-level software. Bridging this gap requires analyses that model realistic low-level C behavior without compromising soundness or scalability.

One of the most established tools in this area is AProVE, introduced by Giesl et al. [25] as a platform that transforms programs into term rewrite systems (TRSs) via symbolic execution graphs, enabling rule-based termination proofs using dependency pairs and well-founded orderings. Ströder et al. [33] extended it to support memory-manipulating C programs through symbolic execution with memory safety analysis, enabling reasoning over pointer arithmetic and heap operations. Hensel et al. [50] added non-termination analysis, using symbolic execution and SMT solving (Satisfiability Modulo Theories) to detect infinite executions involving recursion and dynamic memory. AProVE also supports fixed-width bit-vector arithmetic via modular integer representations [34]. However, its abstractions—based on TRSs and integer transition systems (ITSs)—rely on over-approximated integers, limiting precision for bit-level operations. It also lacks support for complex memory layouts like structs and unions, and does not preprocess pointer-heavy or structurally intricate constructs, reducing its applicability to low-level C.

Another major line of work is UAutomizer, a model checker in the Ultimate framework [28, 29]. It verifies safety and liveness properties, including termination, using trace abstraction (a path-based technique), Büchi automata (to capture infinite executions), and CEGAR (Counterexample-Guided Abstraction Refinement). Later extensions added array interpolation [31], enabling partial memory

reasoning via array-index abstraction. While UAutomizer detects some memory safety issues—such as null dereferences or invalid frees—it lacks full support for pointer arithmetic, aliasing, and complex layouts like nested structs and unions. Bitwise operations and finite-width arithmetic are over-approximated by default and handled precisely only during refinement [30], limiting accuracy for hardware-sensitive code. It also lacks preprocessing to normalize structurally intricate memory constructs. Thus, although effective on control-heavy programs with moderate memory use, UAutomizer struggles with low-level C featuring rich data and bit-level semantics.

2LS, introduced by Schrammel and Kroening [48], builds on the CProver infrastructure and combines symbolic execution with template-based invariant inference to verify memory safety and program properties. Kaiser et al. [42] incorporated heap modeling and non-termination precondition inference; later work [15] added bit-precise arithmetic, procedure-modular summaries, and lexicographic ranking functions. Chen et al. [14] proposed a two-phase interprocedural method combining over- and under-approximation to synthesize context-sensitive termination preconditions. Although 2LS is expressive and precise, it does not simplify structurally complex memory constructs before analysis. In contrast, our approach applies targeted rewrites—such as pointer-to-array conversion and finite-width arithmetic instrumentation—that expose analyzable patterns for MuVal's μCLP-based engine.

Proton [43, 44] is a competition-oriented framework for termination and non-termination analysis of C programs. It employs recurrent-state instrumentation combined with bounded model checking to detect non-termination, and applies heuristic and LLM-assisted ranking-function synthesis for termination. However, its reliance on bounded unwinding and lightweight validation limits formal soundness guarantees and bit-precise reasoning for low-level C programs. EndWatch [61] is a hybrid method for detecting non-termination in real-world C programs. It uses symbolic reasoning for linear loops and instrumentation-based fuzzing with state-revisit detection for nonlinear ones. EndWatch uncovered real-world bugs, including previously unknown Common Vulnerabilities and Exposures (CVEs), but is inherently incomplete and unsound, relying on runtime behavior and test coverage. It also lacks support for bit-level operations, pointer aliasing, and modular arithmetic. Pulse$^\infty$ [47] is a compositional framework for heap-aware non-termination analysis based on UNTerSL, a separation logic for under-approximating divergence. Pulse$^\infty$ scales to large C codebases and was evaluated on over 2,000 functions from major open-source systems. While it provides precise reasoning about dynamic memory and recursive heaps, it does not model finite-width arithmetic or bit-level semantics, and lacks preprocessing of low-level constructs.

4 Approach

This section introduces Athena, a framework for sound termination and non-termination analysis of C programs that models finite-width and bit-precise integer semantics and supports advanced constructs such as pointers, arrays, structures, and bitwise operations.

4.1 Overview of Athena

Figure 2 shows the analysis pipeline of Athena. Given a C program, Athena first rewrites pointer-based memory operations into array indexing, yielding structured memory access suitable for array-based termination analysis and simplifying subsequent reasoning. The program then passes through the bounded-integer semantics assurance engine, where finite-width integer semantics may be enforced. Three modes are supported: (1) *None*, which passes the array-based program directly for analysis under mathematical integer semantics—for example, in benchmarks like TermCOMP that assume this semantics, or after CPAchecker confirms that no finite-width overflow/underflow can occur as a fallback; (2) *Modulo Arithmetic (MA)*, which instruments operations with wraparound to model finite-width arithmetic effects; and (3) *Bit-Vector Semantics (BV)*, which extracts type

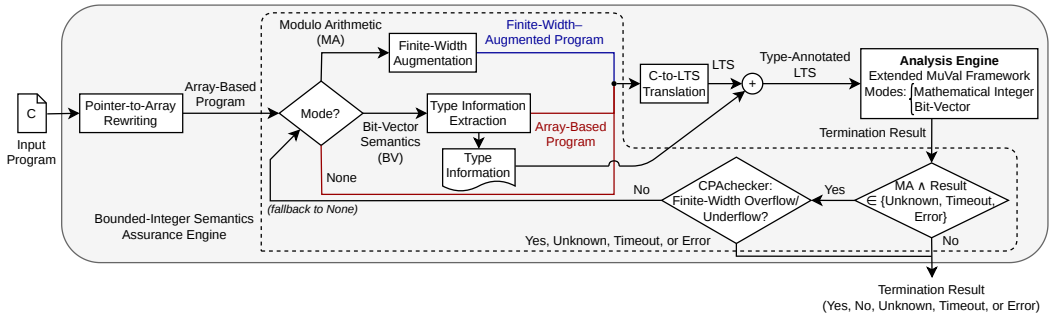


Fig. 2. Overview of the Athena analysis framework.

information (widths and signedness) and annotates it on the Labeled Transition System (LTS). The result is either a plain array-based program or a finite-width–augmented one, which is translated into an LTS by our extended llvm2KITTeL supporting arrays, structures, and bitwise arithmetic. The LTS is then analyzed by an extended MuVal, which integrates array, tuple, and bit-vector theories for ranking function synthesis and recurrent set detection. Depending on the chosen mode, reasoning is performed in the theory of mathematical integers (for None and MA) or bit-vectors (for BV). In both cases, it enables precise analysis of complex C programs with one of the outcomes: terminating, non-terminating, unknown, timeout, or error. If analysis under the modulo arithmetic mode is inconclusive, Athena invokes CPAchecker to check whether any finite-width overflow/underflow can occur and, when none is possible, the array-based program is re-analyzed in None mode, ensuring soundness without unnecessary overhead.

4.2 Pointer-to-Array Rewriting

Athena introduces a pointer-to-array rewriting transformation that systematically converts pointer-based memory operations into structured array indexing. Pointers in C exhibit complex behaviors—such as arithmetic, aliasing, casting, and indirect memory access—that are difficult to model precisely and soundly in logic-based frameworks [59]. Consequently, many verification tools either lack support for pointer-intensive programs altogether, restrict pointer usage, or employ unsound approximations [36, 59]. Modern SMT solvers provide mature theories for arrays and bit-vectors, but lack an equally strong, standardized theory for heap pointers and arbitrary pointer arithmetic. Encoding pointer semantics directly in a μCLP-style solver would therefore require a rich symbolic heap and substantial back-end extensions. To avoid this, our transformation replaces pointer operations with array-based memory accesses, allowing the back end to rely on standard array theories and enabling compatibility with tools that support arrays but not low-level pointer constructs. By making memory accesses explicit and analyzable, it facilitates precise reasoning about complex memory interactions [36, 59]. The transformation proceeds in three phases: (1) pointer abstraction, (2) memory block construction, and (3) rewriting of pointer constructs, resulting in a semantically equivalent (w.r.t. the target fragment), pointer-free program suitable for array-based analysis. We first make explicit the target C fragment for which the rewriting rules are defined, and then present the three phases of the transformation.

4.2.1 Target C Fragment. Athena's pointer-to-array transformation is defined for a C fragment that includes integral scalars and arrays, user-defined `struct` types, and first-order functions whose parameters and return values may be pointers. Within this fragment, we support pointers of arbitrary depth; pointer casts; address-of and dereference; offset-based memory access via array

```
1  extern int __VERIFIER_nondet_int(void);
2  typedef struct Node {
3    int value; struct Node* next;
4  } Node;
5  Node* process(Node* n, char c) {
6    if (c == 'A') return n->next;
7    return NULL;
8  }
9  int main() {
10   int x = __VERIFIER_nondet_int();
11   int y = __VERIFIER_nondet_int();
12   int z = __VERIFIER_nondet_int();
13   int arr[10];
14   for (int i = 0; i < 10; ++i)
15     arr[i] = __VERIFIER_nondet_int();
16   int* p1 = &x;
17   int* p2 = &y;
18   char* p3 = (char*)&y;
19   int* p4 = arr;
20   int* p5 = arr[5] <= y ? &x : &z;
21   int** p6 = *p5 % 2 == 0 ? &p1 : &p4;
22   int** p7 = &p5;
23   Node* head = (Node*)malloc(sizeof(Node));
24   head->value = 0;
25   head->next = head;
26   while (**p6 < *p2) {
27     **p6 += **p7;
28     if (process(head, *p3) != NULL)
29       **p6 -= **p7;
30   }
31   return 0;
32 }
```

(a) Pointer-based program.

```
1  extern int __VERIFIER_nondet_int(void);
2  typedef struct Node {
3    int value; int next;
4  } Node;
   // Declaration of memory blocks and
        memory management functions
5  int process(int n, char c) {
6    if (c == 'A') return m5[n].next;
7    return 0;
8  }
9  int main() {
10   int x_addr = alloc_m1(1);
11   m1[x_addr] = __VERIFIER_nondet_int();
12   int y_addr = alloc_m2(sizeof(int));
13   store_m2_int(y_addr,
          __VERIFIER_nondet_int());
14   int z_addr = alloc_m1(1);
15   m1[z_addr] = __VERIFIER_nondet_int();
16   int arr_addr = alloc_m1(10);
17   for (int i = 0; i < 10; ++i)
18     m1[arr_addr + i] =
            __VERIFIER_nondet_int();
19   int p1_addr = alloc_m3(1);
20   m3[p1_addr] = x_addr;
21   int p2 = y_addr;
22   int p3 = y_addr;
23   int p4_addr = alloc_m3(1);
24   m3[p4_addr] = arr_addr;
25   int p5_addr = alloc_m4(1);
26   m4[p5_addr] = m1[arr_addr + 5] <=
          load_m2_int(y_addr) ? x_addr : z_addr;
27   int p6 = m1[m4[p5_addr]] % 2 == 0 ? p1_addr
          : p4_addr;
28   int p7 = p5_addr;
29   int head = alloc_m5(1);
30   m5[head].value = 0;
31   m5[head].next = head;
32   while (m1[m3[p6]] < m2[p2]) {
33     m1[m3[p6]] += m1[m4[p7]];
34     if (process(head, load_m2_char(p3)) != 0)
35       m1[m3[p6]] -= m1[m4[p7]];
36   }
37   return 0;
38 }
```

(b) Array-based program.

Fig. 3. Pointer-based program and equivalent array-based form via Athena's pointer-to-array rewriting.

indexing and pointer arithmetic; structure-field access through pointers; and dynamic allocation via `alloca` and `malloc`. Figure 3 illustrates a representative program exercising these constructs, and the rules in Figure 5 define the rewriting for this fragment, with each rule corresponding to a distinct class of supported constructs.

We treat explicit deallocation via `free` and `union` types as out of scope: `free` introduces dynamic lifetimes not modeled by our pointer-to-array abstraction, while `union` yields overlapping storage that may be lowered in LLVM IR using low-level encodings (e.g., `bitcast`), which are not currently supported by our llvm2KITTeL-based C-to-LTS translation (Section 4.4) without additional abstraction. The C-to-LTS translation further relies on function inlining; consequently, general recursion is conservatively treated as out of scope end-to-end, as discussed in Section 4.4.

4.2.2 Pointer Abstraction. The rewriting process starts by extracting a high-level abstraction of pointer variables, capturing each pointer's type and memory relationships. This abstraction forms the basis for constructing memory blocks and rewriting pointer constructs.

We formalize the pointer abstraction as a set $\mathcal{A} = \{(p_i, \tau_i, Q_i) \mid 1 \leq i \leq n\}$, where n is the total number of pointer variables, and each tuple encodes the following information:

- p_i: the i-th pointer variable in the program.
- τ_i: the type pointed to by p_i, recursively defined as $\tau ::= \sigma \mid \tau^*$, where σ is a base type (e.g., `char`, `int`, or a user-defined `struct`) and τ^* is a pointer to τ, enabling arbitrarily nested pointer types.
- $Q_i = \{q_{ij} \mid 1 \leq j \leq m\}$: the points-to set of p_i, i.e., the set of memory locations it may reference, including statically allocated variables, other pointers, and heap objects distinguished by allocation site. The value $m = |Q_i|$ denotes the number of such locations. We derive Q_i using DG [11], an allocation-site–based may-points-to analysis in a flow-sensitive, context-insensitive, and field-insensitive setting over LLVM IR. DG propagates may-alias information across function boundaries (including pointer parameters), and distinguishes heap objects by allocation site.

Consider Figure 3a, which shows a C program with representative pointer constructs, including single- and multi-level pointers, aliasing, casting, and dynamically allocated structures, as well as a non-terminating loop. This motivating example illustrates the generality of our transformation process. From its pointer analysis, we extract the following abstraction:

$$\mathcal{A} = \left\{ \begin{array}{l} (p1, \texttt{int}, \{x\}), (p2, \texttt{int}, \{y\}), (p3, \texttt{char}, \{y\}), (p4, \texttt{int}, \{arr\}), (p5, \texttt{int}, \{x, z\}), \\ (p6, \texttt{int*}, \{p1, p4\}), (p7, \texttt{int*}, \{p5\}), (head, \texttt{Node}, \{\texttt{malloc}\}), (n, \texttt{Node}, \{\texttt{malloc}\}) \end{array} \right\}$$

For example, $(p1, \texttt{int}, \{x\})$ indicates that $p1$ may reference the integer variable x, whereas $(p6, \texttt{int*}, \{p1, p4\})$ identifies $p6$ as a second-level pointer that may reference either $p1$ or $p4$.

4.2.3 Memory Block Construction. Given the pointer abstraction $\mathcal{A}$, we construct a finite set of disjoint memory blocks, each grouping semantically related pointer variables with overlapping target locations to form the basis for rewriting indirect memory accesses into analyzable array indexing.

We define the set of memory blocks as $\mathcal{M} = \{\langle m_i,\ \sigma_i\rangle \mapsto (P_i, T_i, Q_i) \mid i \in \mathbb{N}^+\}$, where:

- m_i: symbolic identifier of the i-th memory block.
- σ_i: data type assigned to memory block m_i.
- P_i: set of pointer variables grouped into memory block m_i.
- T_i: set of types pointed to by pointers in P_i.
- Q_i: union of the points-to sets of all pointers in P_i.

The set $\mathcal{M}$ is constructed by Algorithm 1, which iterates over each $(p_i, \tau_i, Q_i) \in \mathcal{A}$ and incrementally builds memory blocks. A pointer joins an existing block if its points-to set intersects with that of the block or if it aliases another pointer in the block (i.e., they co-occur in a points-to set); otherwise, a new block is created. After grouping, the algorithm assigns each block a type σ_i based on T_i. If any type in T_i is a (possibly nested) pointer, σ_i is set to `int`, as pointers are mapped to integer indices. If T_i is a singleton non-pointer, non-structure type, that type is used. If T_i contains multiple types or a structure type with address-taken fields, σ_i defaults to `unsigned char` to preserve byte-level access.

Applying this procedure to the abstraction $\mathcal{A}$ from Figure 3a yields the following memory blocks:

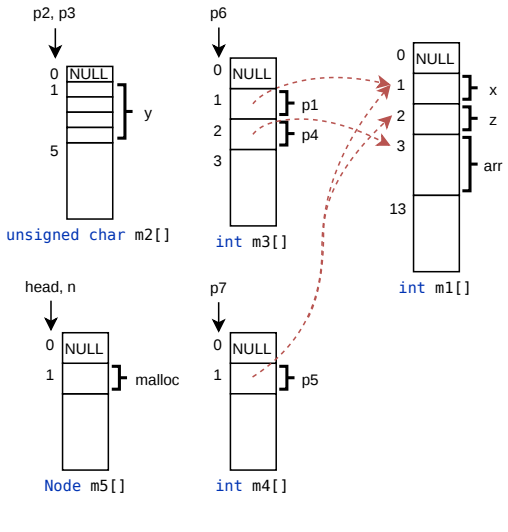
$$\mathcal{M} = \left\{ \begin{array}{l} \langle m_1, \texttt{int}\rangle \mapsto (\{p1, p4, p5\}, \{\texttt{int}\}, \{x, z, arr\}), \\ \langle m_2, \texttt{unsigned char}\rangle \mapsto (\{p2, p3\}, \{\texttt{int}, \texttt{char}\}, \{y\}), \\ \langle m_3, \texttt{int}\rangle \mapsto (\{p6\}, \{\texttt{int*}\}, \{p1, p4\}), \\ \langle m_4, \texttt{int}\rangle \mapsto (\{p7\}, \{\texttt{int*}\}, \{p5\}), \\ \langle m_5, \texttt{Node}\rangle \mapsto (\{head, n\}, \{\texttt{Node}\}, \{\texttt{malloc}\}) \end{array} \right\}$$

These blocks are visually illustrated in Figure 4. Each m_i denotes a distinct memory block of type σ_i, storing all targets in Q_i, with each pointer in P_i serving as an access index into the block. For example, m_2, of type `unsigned char`, stores the integer variable y, accessed via $p2$ and $p3$.

Algorithm 1: Memory Block Construction.

Input: $\mathcal{A} = \{(p_i, \tau_i, Q_i) \mid 1 \le i \le n\}$, where n is the total number of pointer variables in the program.
Output: $\mathcal{M} = \{\langle m_i,\ \sigma_i\rangle \mapsto (P_i, T_i, Q_i) \mid i \in \mathbb{N}^+\}$

$\mathcal{M} \leftarrow \emptyset$, $counter \leftarrow 1$;
foreach $(p_i, \tau_i, Q_i) \in \mathcal{A}$ **do**
  $merged \leftarrow$ **false**;
  foreach $\langle m_j, \sigma_j\rangle \mapsto (P_j, T_j, Q_j) \in \mathcal{M}$ **do**
    if $Q_i \cap Q_j \neq \emptyset$ ***or*** $\exists Q_k \in \mathcal{A}$ *such that* $\{p_i, p_j\} \subseteq Q_k$ *for some* $p_j \in P_j$ **then**
      $P_j \leftarrow P_j \cup \{p_i\}$, $T_j \leftarrow T_j \cup \{\tau_i\}$, $Q_j \leftarrow Q_j \cup Q_i$;
      $merged \leftarrow$ **true**;
      break;
  if ***not*** $merged$ **then**
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle m_{counter},\ \sigma_{counter}\rangle \mapsto (\{p_i\}, \{\tau_i\}, Q_i)\}$;
    $counter \leftarrow counter + 1$;
foreach $\langle m_i,\ \sigma_i\rangle \mapsto (P_i, T_i, Q_i) \in \mathcal{M}$ **do**
  Let τ_1 be the first element in T_i;
  if τ_1 *is a pointer type* **then** $\sigma_i \leftarrow$ `int`;
  else if τ_1 *is a struct type* ***and*** *address-of operator is applied to its fields* **then** $\sigma_i \leftarrow$ `unsigned char`;
  else if $|T_i| = 1$ **then** $\sigma_i \leftarrow \tau_1$;
  else $\sigma_i \leftarrow$ `unsigned char`;
return $\mathcal{M}$;




```c
#define M2_SIZE 100000
unsigned char m2[M2_SIZE];
int m2_freeIndex = 1;
int alloc_m2(int size) {
    int allocatedIndex = m2_freeIndex;
    m2_freeIndex = m2_freeIndex + size;
    return allocatedIndex;
}
char load_m2_char(int index) {
    return (char)m2[index];
}
void store_m2_char(int index, int value) {
    m2[index] = (unsigned char)value;
}
int load_m2_int(int index) {
    int b0 = m2[index];
    int b1 = m2[index + 1];
    int b2 = m2[index + 2];
    int b3 = m2[index + 3];
    return b0 + b1 * 256 + b2 * 256 * 256 + b3
* 256 * 256 * 256;
}
void store_m2_int(int index, int value) {
    unsigned int uvalue = (unsigned int)value;
    m2[index] = (uvalue / 1) % 256;
    m2[index + 1] = (uvalue / 256) % 256;
    m2[index + 2] = (uvalue / 65536) % 256;
    m2[index + 3] = (uvalue / 16777216) % 256;
}
```

Fig. 4. Memory blocks.

Since these pointers alias y with incompatible types, m_2 requires byte-level access, handled by the `load`/`store` functions, as shown in the figure. In contrast, m_1, of type `int`, stores x, z, and the array `arr`, accessed by $p1$, $p4$, and $p5$. The value at $p5$ (allocated in m_4) holds the index of either x or z in m_1, preserving indirect semantics via index-based access.

R1 - Structure Field Declaration Rewriting:

$$\texttt{struct}\ S\ \{\tau_1\ f_1;\ \ldots;\ \tau_n\ f_n;\} \quad \rightsquigarrow \quad \texttt{struct}\ S\ \{\tau'_1\ f_1;\ \ldots;\ \tau'_n\ f_n;\} \quad \text{where } \tau'_i = \begin{cases} \texttt{int} & \text{if } f_i \in P \\ \tau_i & \text{otherwise} \end{cases} \quad \text{for all } i \in [1..n]$$

R2 - Function Return Type Rewriting:

$$\tau\ f(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n) \quad \rightsquigarrow \quad \begin{cases} \texttt{int}\ f(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n) & \text{if } \tau = \tau'^{*} \\ \tau\ f(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n) & \text{otherwise} \end{cases}$$

R3 - Function Parameter Declaration Rewriting:

$$\tau_{\text{ret}}\ f(\tau_1\ x_1,\ \ldots,\ \tau_n\ x_n) \quad \rightsquigarrow$$

$$\tau_{\text{ret}}\ f(\tau'_1\ x'_1,\ \ldots,\ \tau'_n\ x'_n) \ \text{ where } \tau'_i,\ x'_i = \begin{cases} \texttt{int},\ x_i & \text{if } x_i \in P \wedge M(x_i) = \texttt{null} \\ \texttt{int},\ x_i\texttt{\_addr} & \text{if } x_i \in P \wedge M(x_i) \neq \texttt{null} \\ \texttt{int},\ x_i\texttt{\_addr} & \text{if } x_i \in Q \\ \tau_i,\ x_i & \text{otherwise} \end{cases} \quad \text{for all } i \in [1..n]$$

R4 - Scalar Variable Declaration Rewriting:

$$\tau\ v; \quad \rightsquigarrow \quad \begin{cases} \texttt{int}\ v; & \text{if } v \in P \wedge M(v) = \texttt{null} \\ \texttt{int}\ v\texttt{\_addr};\ v\texttt{\_addr} = \texttt{alloc}_{m_i}(1); & \text{if } v \in P \wedge M(v) = m_i, i \in \mathbb{N}^+ \\ \texttt{int}\ v\texttt{\_addr};\ v\texttt{\_addr} = \texttt{alloc}_{m_j}(1); & \text{if } v \in Q \wedge M(v) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \\ \texttt{int}\ v\texttt{\_addr};\ v\texttt{\_addr} = \texttt{alloc}_{m_j}(\texttt{sizeof}(\sigma)); & \text{if } v \in Q \wedge M(v) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \\ \tau\ v; & \text{otherwise} \end{cases}$$

R5 - Array Declaration Rewriting (Element Type σ):

$$\sigma\ a[n]; \quad \rightsquigarrow \quad \begin{cases} \texttt{int}\ a\texttt{\_addr};\ a\texttt{\_addr} = \texttt{alloc}_{m_j}(n); & \text{if } a \in Q \wedge M(a) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \\ \texttt{int}\ a\texttt{\_addr};\ a\texttt{\_addr} = \texttt{alloc}_{m_j}(n \cdot \texttt{sizeof}(\sigma)); & \text{if } a \in Q \wedge M(a) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \\ \sigma\ a[n]; & \text{otherwise} \end{cases}$$

R6 - Pointer Array Declaration Rewriting (Element Type τ^*):

$$\tau^*\ a[n]; \quad \rightsquigarrow \quad \texttt{int}\ a[n];$$

R7 - Pointer-to-Array Declaration Rewriting:

$$\tau\ (*^m p)[n]; \quad \rightsquigarrow \quad \begin{cases} \texttt{int}\ p; & \text{if } M(p) = \texttt{null} \\ \texttt{int}\ p\texttt{\_addr};\ p\texttt{\_addr} = \texttt{alloc}_{m_i}(1); & \text{if } M(p) = m_i, i \in \mathbb{N}^+ \end{cases}$$

R8 - Null Pointer Assignment Rewriting:

$$p = \texttt{NULL}; \quad \rightsquigarrow \quad \begin{cases} p = 0; & \text{if } M(p) = \texttt{null} \\ m_i[p\texttt{\_addr}] = 0; & \text{if } M(p) = m_i, i \in \mathbb{N}^+ \end{cases}$$

R9 - Dynamic Memory Allocation Rewriting:

$$p = (\tau^*)\texttt{alloca}(n * \texttt{sizeof}(\tau)); \text{ or } p = (\tau^*)\texttt{malloc}(n * \texttt{sizeof}(\tau)); \quad \rightsquigarrow$$

$$\begin{cases} p = \texttt{alloc}_{m_j}(n * 1); & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge \tau = \sigma \\ p = \texttt{alloc}_{m_j}(n * \texttt{sizeof}(\sigma)); & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge \tau = \sigma \\ p = \texttt{alloc}_{m_j}(n * \texttt{sizeof}(\texttt{int})); & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge \tau = \tau^* \\ m_i[p\texttt{\_addr}] = \texttt{alloc}_{m_j}(n * 1); & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge \tau = \sigma \\ m_i[p\texttt{\_addr}] = \texttt{alloc}_{m_j}(n * \texttt{sizeof}(\sigma)); & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge \tau = \sigma \\ m_i[p\texttt{\_addr}] = \texttt{alloc}_{m_j}(n * \texttt{sizeof}(\texttt{int})); & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge \tau = \tau^* \end{cases}$$

R10a - Load Rewriting for Scalar Variables:

$$v \quad \rightsquigarrow \quad \begin{cases} v & \text{if } v \in P \wedge M(v) = \texttt{null} \\ m_i[v\texttt{\_addr}] & \text{if } v \in P \wedge M(v) = m_i, i \in \mathbb{N}^+ \\ m_j[v\texttt{\_addr}] & \text{if } v \in Q \wedge M(v) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(v) = \sigma \\ \texttt{load}_{m_j,\,\sigma}(v\texttt{\_addr}) & \text{if } v \in Q \wedge M(v) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(v) = \sigma \\ v & \text{otherwise} \end{cases}$$

R10b - Store Rewriting for Scalar Variables:

$$v = expr; \quad \rightsquigarrow \quad \begin{cases} v = expr; & \text{if } v \in P \wedge M(v) = \texttt{null} \\ m_i[v\texttt{\_addr}] = expr; & \text{if } v \in P \wedge M(v) = m_i, i \in \mathbb{N}^+ \\ m_j[v\texttt{\_addr}] = expr; & \text{if } v \in Q \wedge M(v) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(expr) = \sigma \\ \texttt{store}_{m_j,\,\sigma}(v\texttt{\_addr},\ expr); & \text{if } v \in Q \wedge M(v) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(expr) = \sigma \\ v = expr; & \text{otherwise} \end{cases}$$

R11a - Load Rewriting for Array Elements:

$$a[k] \quad \rightsquigarrow$$

$$\begin{cases} m_j[a + k] & \text{if } a \in P \wedge M(a) = \texttt{null} \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ \texttt{load}_{m_j,\,\sigma}(a + k \cdot \texttt{sizeof}(\sigma)) & \text{if } a \in P \wedge M(a) = \texttt{null} \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ m_j[m_i[a\texttt{\_addr}] + k] & \text{if } a \in P \wedge M(a) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ \texttt{load}_{m_j,\,\sigma}(m_i[a\texttt{\_addr}] + k \cdot \texttt{sizeof}(\sigma)) & \text{if } a \in P \wedge M(a) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ m_j[a\texttt{\_addr} + k] & \text{if } a \in Q \wedge M(a) = m_j,\ j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ \texttt{load}_{m_j,\,\sigma}(a\texttt{\_addr} + k \cdot \texttt{sizeof}(\sigma)) & \text{if } a \in Q \wedge M(a) = m_j,\ j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ a[k] & \text{otherwise} \end{cases}$$

Continued on the next page

R11b - Store Rewriting for Array Elements:

$a[k] = expr;$ $\rightsquigarrow$

$$\begin{cases} m_j[a+k] = expr; & \text{if } a \in P \wedge M(a) = \texttt{null} \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ \texttt{store}_{m_j,\sigma}(a + k \cdot \texttt{sizeof}(\sigma),\ expr); & \text{if } a \in P \wedge M(a) = \texttt{null} \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ m_j[m_i[a\_\texttt{addr}] + k] = expr; & \text{if } a \in P \wedge M(a) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ \texttt{store}_{m_j,\sigma}(m_i[a\_\texttt{addr}] + k \cdot \texttt{sizeof}(\sigma),\ expr); & \text{if } a \in P \wedge M(a) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ m_j[a\_\texttt{addr} + k] = expr; & \text{if } a \in Q \wedge M(a) = m_j,\ j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ \texttt{store}_{m_j,\sigma}(a\_\texttt{addr} + k \cdot \texttt{sizeof}(\sigma),\ expr); & \text{if } a \in Q \wedge M(a) = m_j,\ j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ a[k] = expr; & \text{otherwise} \end{cases}$$

R12a - Load Rewriting for Structure Field Access via Pointer:

$p \rightarrow f$ $\rightsquigarrow$

$$\begin{cases} m_j[p].f & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(f) = \sigma \\ \texttt{load}_{m_j,\sigma}(p + \texttt{offset}(f)) & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(f) = \sigma \\ m_j[m_i[p\_\texttt{addr}]].f & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(f) = \sigma \\ \texttt{load}_{m_j,\sigma}(m_i[p\_\texttt{addr}] + \texttt{offset}(f)) & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(f) = \sigma \end{cases}$$

R12b - Store Rewriting for Structure Field Access via Pointer:

$p \rightarrow f = expr;$ $\rightsquigarrow$

$$\begin{cases} m_j[p].f = expr; & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(f) = \sigma \\ \texttt{store}_{m_j,\sigma}(p + \texttt{offset}(f),\ expr); & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(f) = \sigma \\ m_j[m_i[p\_\texttt{addr}]].f = expr; & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(f) = \sigma \\ \texttt{store}_{m_j,\sigma}(m_i[p\_\texttt{addr}] + \texttt{offset}(f),\ expr); & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(f) = \sigma \end{cases}$$

$\text{offset}(f_i) = \sum_{j=1}^{i-1} \text{sizeof}(f_j)$, where f_i is the i-th field in the structure and f_j are the preceding fields.

R13a - Load Rewriting for Dereferenced Pointer Expression:

$*(p \pm k)$ $\rightsquigarrow$

$$\begin{cases} m_j[p \pm k] & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(*(p \pm k)) = \sigma \\ \texttt{load}_{m_j,\sigma}(p \pm k \cdot \texttt{sizeof}(\sigma)) & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(*(p \pm k)) = \sigma \\ m_j[m_i[p\_\texttt{addr}] \pm k] & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(*(p \pm k)) = \sigma \\ \texttt{load}_{m_j,\sigma}(m_i[p\_\texttt{addr}] \pm k \cdot \texttt{sizeof}(\sigma)) & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(*(p \pm k)) = \sigma \end{cases}$$

R13b - Store Rewriting for Dereferenced Pointer Expression:

$*(p \pm k) = expr;$ $\rightsquigarrow$

$$\begin{cases} m_j[p \pm k] = expr; & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(*(p \pm k)) = \sigma \\ \texttt{store}_{m_j,\sigma}(p \pm k \cdot \texttt{sizeof}(\sigma), expr) & \text{if } M(p) = \texttt{null} \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(*(p \pm k)) = \sigma \\ m_j[m_i[p\_\texttt{addr}] \pm k] = expr; & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(*(p \pm k)) = \sigma \\ \texttt{store}_{m_j,\sigma}(m_i[p\_\texttt{addr}] \pm k \cdot \texttt{sizeof}(\sigma), expr) & \text{if } M(p) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(p)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(*(p \pm k)) = \sigma \end{cases}$$

R14 - Address-of Operation Rewriting for Scalar Variables:

$\&v \rightsquigarrow v\_\texttt{addr}$

R15 - Address-of Operation Rewriting for Array Elements:

$\&a[k]$ $\rightsquigarrow$

$$\begin{cases} a + k & \text{if } a \in P \wedge M(a) = \texttt{null} \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ a + k \cdot \texttt{sizeof}(\sigma) & \text{if } a \in P \wedge M(a) = \texttt{null} \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ m_i[a\_\texttt{addr}] + k & \text{if } a \in P \wedge M(a) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ m_i[a\_\texttt{addr}] + k \cdot \texttt{sizeof}(\sigma) & \text{if } a \in P \wedge M(a) = m_i,\ i \in \mathbb{N}^+ \wedge M(Q(a)) = m_j, j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \\ a\_\texttt{addr} + k & \text{if } a \in Q \wedge M(a) = m_j,\ j \in \mathbb{N}^+ \wedge T(m_j) = \sigma \wedge T(a[k]) = \sigma \\ a\_\texttt{addr} + k \cdot \texttt{sizeof}(\sigma) & \text{if } a \in Q \wedge M(a) = m_j,\ j \in \mathbb{N}^+ \wedge T(m_j) \neq \sigma \wedge T(a[k]) = \sigma \end{cases}$$

Fig. 5. Rewriting rules.

4.2.4 Rewriting of Pointer Constructs. The final phase rewrites all pointer constructs into semantically equivalent (w.r.t. the target fragment) array operations, preserving the original behavior while producing a pointer-free program compatible with array-based analysis tools. The rewriting operates on the program's abstract syntax tree (AST), built using the Clang compiler frontend [54], and recursively applies the syntax-directed rules defined in Figure 5, guided by the memory mapping $\mathcal{M}$, to replace pointer constructs with array-based equivalents.

Figure 5 comprises 15 transformation rules (R1–R15), each handling a specific class of pointer-related constructs, such as dynamic memory allocation, dereferencing, address-of expressions, and

pointer-based access to structures and arrays. The rewriting relies on the following helper sets and functions to ensure semantic and type correctness:

- P: set of all pointer variables, including structure fields of pointer type.
- Q: set of all variables that may be pointed to by any pointer in P.
- $T(x)$: base type of variable x.
- $Q(x)$: points-to set of pointer x.
- $M(x)$: memory block storing x—`null` if direct accessible, or m_i if stored in memory block m_i.

To illustrate the rewriting process, consider the pointer-based program in Figure 3a. After deriving the memory mapping $\mathcal{M}$, each line is transformed to produce the array-based version shown in Figure 3b. Specifically, lines 13–15 in Figure 3a, which declare and initialize the integer array *arr*, are rewritten as lines 16–18 in Figure 3b using Rules R4 and R10b. Line 20, which conditionally assigns to pointer $p5$ the address of either x or z, is transformed into lines 25–26 via a composition of Rules R4, R11a, and R14.

4.3 Bounded-Integer Semantics Assurance

Many real-world C programs rely on low-level arithmetic whose finite-width wraparound and boundary effects may alter control flow and cause non-termination. Many existing termination and non-termination analysis methods assume unbounded integers, abstracting away these effects and risking unsoundness. To address this, Athena allows the user to decide whether bounded integer semantics must be enforced and, if so, which semantic mode is applied. As illustrated in Figure 2, Athena supports three modes: (1) None, (2) Modulo Arithmetic, and (3) Bit-Vector Semantics. The None mode forwards the program for analysis under mathematical integer semantics and is sound only when finite-width effects are irrelevant (e.g., in TermCOMP) or have been ruled out (e.g., after CPAchecker confirms that no finite-width overflow/underflow can occur). The latter two modes enforce bounded integer semantics and are described in the following subsections.

4.3.1 Modulo Arithmetic Mode. In this mode, Athena applies a source-to-source transformation that explicitly embeds bounded integer semantics into the program prior to translation into the Labeled Transition System (LTS).

Consider the motivating example `Type_Conversion_in_Comparison_1_NT` from Figure 1b, which exhibits non-termination due to finite-width wraparound effects in the presence of integer promotions. To capture this behavior, Athena introduces a `wrap_around` helper function shown below, which simulates finite-width wraparound using modular arithmetic. Parameterized by lower and upper bounds, it supports various C integral types and is applied to expressions that may wrap around, identified via AST traversal using Clang's compiler frontend. For example, in Figure 1b, Athena promotes `s`, `seqnum`, and `len` to `long long`, rewrites loop guard `s < seqnum + len` as `s < wrap_around(seqnum + len, INT_MIN, INT_MAX)`, and replaces `s++` with `s = wrap_around(s + 1, 0, USHRT_MAX)`.

```
1  long long wrap_around(long long value, long long lower_bound, long long upper_bound) {
2    long long range = upper_bound - lower_bound + 1;
3    if (value > upper_bound)
4      return lower_bound + (value - upper_bound - 1) % range;
5    else if (value < lower_bound)
6      return upper_bound - (lower_bound - value - 1) % range;
7    return value;
8  }
```

This mode allows the analysis engine to reason in the simpler theory of mathematical integers while preserving the effects of bounded arithmetic. If analysis under this mode is inconclusive, Athena invokes CPAchecker to check whether any finite-width overflow/underflow can occur.

When no such finite-width effect is possible, the wraparound instrumentation is discarded, and the program is re-analyzed in the theory of mathematical integers, thereby ensuring sound results without incurring unnecessary semantic overhead.

4.3.2 Bit-Vector Semantics Mode. In this mode, the program remains unmodified at the source level, and bounded integer semantics are enforced directly during analysis. A dedicated type information extraction pass records variable widths and operator signedness, attaching this metadata to the LTS during its construction. This mode allows the analysis engine to reason directly in the theory of bit-vectors, thereby capturing finite-width, bit-precise integer semantics without requiring source-level rewriting.

4.4 C-to-LTS Translation

In this step, the input program—already transformed into a pointer-free form and optionally finite-width–augmented according to the selected mode—is translated into a Labeled Transition System (LTS), which serves as input to the analysis engine. The program is first compiled to LLVM Intermediate Representation (LLVM-IR) using the LLVM framework, then translated into an abstract LTS representation via llvm2KITTeL. A key contribution of our work is extending llvm2KITTeL to support constructs common in real-world C programs, including arrays, structures, and bit-level operations. Pointers are omitted at this stage, as they are eliminated during the pointer-to-array rewriting phase. In bit-vector mode, the extracted type metadata (bit-widths and signedness) is preserved during translation and determines whether operations and comparisons are encoded using signed or unsigned bit-vector semantics.

The LTS representation supports the following abstract operations:

Array Operations.

- `a := nondet();` declares a nondeterministic array `a`.
- `a := const_array(v);` declares an array `a` with all entries set to `v`.
- `v := select_array(a, i);` reads the value at index `i` from array `a`.
- `a := store_array(a, i, v);` writes `v` to index `i` in array `a`.

Structure Operations.

- `t := constr_tuple(f1, ..., fn);` creates a tuple `t` with fields `f1` to `fn`.
- `v := select_tuple(t, i, n);` extracts the `i`-th field from an `n`-field tuple `t`.

Bit-Level Operations.

- `TYPE v: bv(w);` declares `v` as a bit-vector of width `w`.
- `v := and(x, y);`, `v := or(x, y);`, `v := shl(x, y);`, `v := lshr(x, y);`, and `v := ashr(x, y);` perform standard bitwise logical and shift operations, where `x` is the input value and `y` specifies either the second operand (for logical operations) or the shift amount (for shift operations).
- `v := sign_extend(w_src, w_dst, x);` and `v := zero_extend(w_src, w_dst, x);` both extend the bit-width of `x` from `w_src` to `w_dst`, the former replicating the most significant (sign) bit and the latter padding zeros on the left.
- `v := extract(w_hi, w_lo, x);` extracts the bits from index `w_hi` down to `w_lo` (inclusive) from `x`, effectively truncating or slicing the bit-vector.

Floating-Point Operations.

- `v := int2real(x);` and `v := real2int(x);` convert between integer and floating-point (real) values, with `real2int` discarding the fractional part of `x`.

Figure 6 shows a C program (a) and its corresponding LTS (b). The program defines a `Device` structure, creates an array of such structures, and conditionally updates fields using bitwise logic. In the LTS, our newly introduced abstractions appear explicitly: array and structure operations

```
1  extern int __VERIFIER_nondet_int(void);
2  struct Device { int id; int isOn; };
3  int main() {
4    struct Device devices[2];
5    devices[1].isOn = __VERIFIER_nondet_int() % 2;
6    while ((devices[1].isOn & 1) == 1)
7      devices[0].isOn = 0;
8    return 0;
9  }
```

(a) C program.

```
1  START: main_bb0;
2  TYPE v6: bv(32);
3  FROM: main_bb0;
4  vdevices := nondet();
5  v0 := nondet();
6  v1 := v0 % 2;
7  v2 := select_array(vdevices,1);
8  v3v0 := select_tuple(v2,0,2);
9  v3v2 := constr_tuple(v3v0,v1);
10 vdevices := store_array(vdevices,1,v3v2);
11 v4 := select_array(vdevices,1);
12 v6 := select_tuple(v4,1,2);
13 v7 := and(v6,1);
14 v8 := select_array(vdevices,0);
15 TO: main_bb1;
16 FROM: main_bb1;
17 TO: main_bb1_end;
18 FROM: main_bb1_end;
19 assume(v7 == 1);
20 TO: main_bb2;
21 FROM: main_bb1_end;
22 assume(v7 != 1);
23 TO: main_bb3;
24 FROM: main_bb2;
25 v9v0 := select_tuple(v8,0,2);
26 v9v2 := constr_tuple(v9v0,0);
27 vdevices := store_array(vdevices,0,v9v2);
28 TO: main_bb1;
29 FROM: main_bb3;
30 TO: main_bb3_ret;
```

(b) LTS Representation.

Fig. 6. C program and its LTS representation.

are represented using `select_array`, `store_array`, `constr_tuple`, and `select_tuple`, while bit-level reasoning appears in the type annotation `TYPE v6: bv(32);` and in the encoding of `(isOn & 1)` with the and instruction.

The current C-to-LTS translation aggressively inlines functions; consequently, Athena does not support general recursion end-to-end. We automatically rewrite tail recursion into equivalent loops prior to translation, but conservatively return Unknown for non-tail recursion. Supporting general recursion would require a call-preserving LLVM-to-LTS encoding with an explicit stack/frame model (including frame-local state and pointer relationships), which we leave as future work.

4.5 Termination and Non-termination Analysis

Athena builds on an extended version of MuVal, a modular verification framework based on μCLP, a first-order fixpoint logic with background theories. MuVal performs termination and non-termination analysis using a primal-dual fixpoint computation that combines inductive (safety/termination) and coinductive (liveness/non-termination) reasoning. It synthesizes ranking functions to prove termination and constructs recurrent sets to witness non-termination. However, the original MuVal lacks native support for arrays, tuples, and bit-vectors, limiting its applicability to real-world C programs. We address this by extending MuVal with support for these theories, enabling sound and precise reasoning over μCLP formulas derived from the LTS representation.

To realize this, we extended MuVal's backend solver, PCSat, to support reasoning over array, tuple, and bit-vector theories. These extensions required augmenting PCSat's constraint language and constraint solving engine to interpret and manipulate formulas involving finite-width, bit-precise integer arithmetic and indexed memory. On top of this, we implemented template-based synthesis mechanisms that extend the relational verification templates introduced in PCSat's original design [56], enabling the generation of inductive invariants, Skolem functions, and ranking functions over the newly supported theories. The design of these templates was guided by studying the structure of correctness certificates used in CHC-COMP [12], TermCOMP, and the benchmarks of Shi et al., with the goal of balancing expressiveness and automation. While the specific templates

are not detailed here, they were crafted to accommodate the diverse reasoning patterns observed in practical termination and non-termination proofs. These enhancements integrate into MuVal's existing primal-dual fixpoint strategy, preserving its modular architecture while significantly broadening its applicability to real-world low-level programs.

4.6 Soundness Argument

We argue soundness compositionally across four layers: (1) pointer-to-array rewriting, (2) the selected integer-semantics mode, (3) C-to-LTS translation, and (4) MuVal's reasoning over the translated system. End-to-end soundness follows by composition.

Preliminaries. Let P range over sequential C programs in the target C fragment (Section 4.2.1), and consider only executions with defined behavior under that fragment's semantics. Let σ range over initial program states. For each integer-semantics mode $\mu \in \{\mathsf{None}, \mathsf{Modulo}, \mathsf{BitVector}\}$, we write $\langle P, \sigma\rangle \Downarrow^{\mu}$ (resp. $\langle P, \sigma\rangle \Uparrow^{\mu}$) to denote that the execution of P from initial state σ terminates (resp. diverges) under the integer semantics of μ. We write $P \Downarrow^{\mu}$ iff $\forall\sigma.\ \langle P, \sigma\rangle \Downarrow^{\mu}$, and $P \Uparrow^{\mu}$ iff $\exists\sigma.\ \langle P, \sigma\rangle \Uparrow^{\mu}$. Let $\mathsf{PA}(P)$ denote DG's sound may-alias over-approximation, and let $P' \triangleq \mathsf{Rewrite}(P, \mathsf{PA}(P))$ be the pointer-free program produced by Athena. Fix an integer-semantics mode μ, and let P'_{μ} denote the program obtained by applying the arithmetic semantics of μ in our pipeline. Let $T_{\mu} \triangleq \mathsf{LTS}(P'_{\mu})$ be the LTS produced by our llvm2KITTeL-based C-to-LTS translation. Finally, let $\mathsf{MuVal}(T_{\mu}) \in \{\textsc{Terminates}, \textsc{Nonterminates}, \textsc{Unknown}, \textsc{Timeout}, \textsc{Error}\}$ be MuVal's result.

Lemma 4.1 (Pointer-to-array rewriting preserves termination behavior). *Fix a mode μ. Assume P satisfies the target-fragment restrictions and DG returns a sound may-alias over-approximation $\mathsf{PA}(P)$. Then P and $P' = \mathsf{Rewrite}(P, \mathsf{PA}(P))$ are termination-equivalent under μ, i.e., $P \Downarrow^{\mu} \iff P' \Downarrow^{\mu}$ and $P \Uparrow^{\mu} \iff P' \Uparrow^{\mu}$.*

Proof sketch. The rewriting is justified by a standard memory-abstraction simulation. Each allocation site is mapped to a stable abstract block, realized in P' as a dedicated array (or, when the may-alias information permits accesses through incompatible types, as a byte-level array accessed via typed `load`/`store` helpers that interpret the same underlying bytes according to the accessed type). Dereference and offset/index expressions are rewritten into explicit indexing on the corresponding block. Because $\mathsf{PA}(P)$ is a sound may-alias over-approximation, it may conservatively merge distinct allocation sites in the abstract block partition, potentially reducing solver precision. This impacts only the encoding, not the semantics: no concrete aliasing behavior is excluded and no spurious execution traces are introduced. Therefore, the rewriting preserves the memory semantics of P exactly and is both sound and complete. Under the fixed arithmetic semantics of μ, each rewriting rule preserves the defined read/write effects and induced control-flow decisions, yielding a stepwise correspondence between executions of P and P' and thus termination equivalence. □

Lemma 4.2 (Soundness of integer-semantics modes). *Fix a mode μ. Then termination under μ for P' is equivalent to termination of the encoded program P'_{μ} under the base arithmetic semantics* None, *i.e., $P' \Downarrow^{\mu} \iff P'_{\mu} \Downarrow^{\mathsf{None}}$ and $P' \Uparrow^{\mu} \iff P'_{\mu} \Uparrow^{\mathsf{None}}$.*

Proof sketch. For bounded modes (e.g., modulo or bit-vector semantics), each integer operator is encoded directly in the target theory with its intended meaning; thus, executions of P' under μ correspond exactly to executions of P'_{μ} under None. For mathematical integers, Athena switches to unbounded reasoning only when finite-width effects are irrelevant or CPAchecker proves that no relevant arithmetic operation can overflow or underflow. In the latter case, soundness is conditional on CPAchecker; disabling this fallback yields unconditional soundness at the cost of precision. □

Lemma 4.3 (C-to-LTS translation preserves termination behavior). *Let* $T_\mu \triangleq \mathsf{LTS}(P'_\mu)$. *Then* $P'_\mu \Downarrow^{\mathsf{None}} \iff T_\mu \Downarrow^{\mathsf{None}}$ *and* $P'_\mu \Uparrow^{\mathsf{None}} \iff T_\mu \Uparrow^{\mathsf{None}}$.

Proof sketch. The translation compiles P'_μ to LLVM IR and encodes each LLVM instruction and control-flow edge as LTS transitions. Our llvm2KITTeL extensions provide operational encodings for all supported constructs (Section 4.4), so each defined program step is mirrored by an LTS step with the same state update and control transfer, and vice versa. Therefore, termination and divergence under None semantics are preserved. □

Assumption 1 (MuVal soundness for reported results). *We assume the soundness of MuVal as established in its original work: for any translated system* T_μ *under the supported theories, if* $\mathsf{MuVal}(T_\mu) =$ *Terminates (resp. Nonterminates), then* $T_\mu \Downarrow^{\mathsf{None}}$ *(resp.* $T_\mu \Uparrow^{\mathsf{None}}$*).*

Corollary 4.4 (End-to-end soundness of Athena). *Under the assumptions of Lemmas 4.1–4.3 and Assumption 1 (and, for the unbounded integer option, the CPAchecker side condition when the fallback is enabled), Athena is sound:*

- *If* $\mathsf{MuVal}(T_\mu) =$ *Terminates, then* $P \Downarrow^\mu$.
- *If* $\mathsf{MuVal}(T_\mu) =$ *Nonterminates, then* $P \Uparrow^\mu$.
- *If* $\mathsf{MuVal}(T_\mu) \in \{$*Unknown*, *Timeout*, *Error*$\}$, *Athena makes no claim.*

Proof sketch. Immediate by composition of Lemmas 4.1, 4.2, 4.3, and Assumption 1. □

5 Implementation

Athena is implemented in C++23 on Clang/LLVM 21.1.5. It uses DG to extract flow-sensitive points-to sets from LLVM IR, which are grouped into memory blocks following Algorithm 1. Pointer constructs are identified with Clang's `RecursiveASTVisitor` and rewritten via `clang::Rewriter` according to the rules in Figure 5. For bounded-integer semantics assurance, Athena provides three complementary modes. In None mode, the program is forwarded directly for translation into the LTS. In modulo arithmetic mode, a custom Clang visitor/rewriter instruments arithmetic expressions that may overflow or underflow with calls to the `wrap_around` function (Section 4.3.1). If this analysis yields an inconclusive result, Athena invokes CPAchecker's finite-width overflow/underflow analysis to re-analyze programs confirmed as free of such effects on the relevant operations. In bit-vector mode, Clang extracts type information, which is then attached as annotations to the LTS during construction. All transformations are applied at the source level to preserve compatibility with LLVM compilation.

At the IR level, we extend llvm2KITTeL to support arrays, structures, and bit-level operations. Custom LLVM passes handle constructs such as `alloca`, `getelementptr`, `shl`, and `sitofp`, emitting LTS rules in SMT-compatible syntax. Finally, we extended MuVal's backend solver, PCSat, by augmenting its constraint language and constraint solving engine to support arrays, tuples, and bit-vectors. We added modular, theory-specific template instantiations to enable automated synthesis of ranking functions and recurrent sets. These extensions were benchmark-driven and designed to preserve the solver's primal-dual convergence strategy while maintaining modularity and seamless integration within Athena's analysis pipeline.

6 Evaluation

We evaluate Athena against six state-of-the-art termination analyzers—Proton, AProVE, UAutomizer, CPAchecker, 2LS, and MuVal—on two complementary benchmark suites. The first is the C category of the 2024 Termination Competition (TermCOMP), comprising 828 C programs with pointers and data structures under unbounded integer semantics. The second is the real-world

benchmark of Shi et al. [24, 49], containing 117 C/C++ programs with pointers, arrays, data structures, bitwise operations, and bounded arithmetic, designed to capture the semantic and structural complexity of practical C code. To ensure well-defined semantics, we exclude all benchmarks whose behavior depends on C undefined behavior (UB), as explicitly designated by the benchmark providers—including 12 benchmarks from Shi et al. involving signed integer overflow/underflow and invalid shift operations, and 27 benchmarks from TermCOMP's `AProVE_memory_unsafe` category involving invalid pointer dereference, out-of-bounds memory access, or uninitialized reads—and report results on the remaining programs. Experiments were run on an Apple M2 Pro (10-core, 16 GB RAM, macOS 15.6.1). All tools used the same configurations as Shi et al., with identical timeouts (15 minutes for TermCOMP, 20 minutes for Shi et al.) to ensure fairness. All reported runtimes for Athena include the end-to-end pipeline, including DG pointer analysis, pointer-to-array rewriting, bounded-integer mode selection, C-to-LTS translation, and PCSat solving.

For TermCOMP, Athena was executed in the None mode, using mathematical integer semantics to align with its assumption of unbounded integers. For the Shi et al. benchmarks, which exercise bounded integer semantics, Athena was run in the Bit-Vector (BV) and the Modulo Arithmetic (MA) modes under a split 20-minute budget:

(1) **Athena$_{BV}$:** runs in bit-vector mode for 10 minutes.
(2) **Athena$_{MA}$:** runs in modulo arithmetic mode for 10 minutes. If a decisive result (termination or non-termination) is obtained within half of the time budget, we directly report it with its runtime. Otherwise, CPAchecker checks whether any finite-width overflow/underflow is possible. If none is possible, we conservatively report the initial indecisive outcome (unknown, timeout, or error) with the accumulated runtime. Otherwise, we rerun in the None mode and report that result with the accumulated runtime (original attempt + CPAchecker + rerun).
(3) **Athena (combined):** reports the union of Athena$_{BV}$ and Athena$_{MA}$. If Athena$_{BV}$ produces a decisive result, it is adopted with its runtime; otherwise, Athena$_{MA}$'s result is taken, with aggregated runtimes.

We structure our evaluation around the following research questions:

- **RQ1:** How does Athena compare with existing tools in terms of correctness (both correct and incorrect analyses)?
- **RQ2:** Can Athena handle advanced C constructs such as pointers, arrays, structures, bitwise operations, and bounded integers?
- **RQ3:** Is Athena sound with respect to the selected bounded integer semantics?
- **RQ4:** What is the impact of Athena's semantic and memory-aware enhancements over baseline tools?
- **RQ5:** How does Athena perform in runtime compared to state-of-the-art tools?
- **RQ6:** How do the bit-vector and modulo arithmetic modes complement each other, and what benefits arise from combining them?

Figure 7 summarizes the outcomes for all tools across both benchmarks. Results are classified as *correct* (sound proof or disproof of termination against the ground truth from TermCOMP and Shi et al.), *wrong* (incorrect result), *unknown* (no conclusion), *timeout* (exceeded time limit), and *error* (internal failure). On the Shi et al. benchmarks, Athena achieves 60.95% correctness, outperforming UAutomizer (43.81%) and matching MuVal (60.95%), while trailing Proton (66.67%). Importantly, Athena produces zero wrong results on this benchmark set, whereas Proton and MuVal report 8.57% and 21.90% wrong results, respectively, highlighting Athena's soundness. On TermCOMP, Athena attains 76.28% correctness, surpassing Proton (75.91%)—the SV-COMP 2025 Termination winner [51]—and AProVE 72.91%—the TermCOMP 2024 winner [53]—while ranking just below UAutomizer 78.65%, the runner-up in TermCOMP 2024 and SV-COMP 2025 (Termination). This gap stems from (1) UAutomizer's stronger recursion handling, (2) our current pointer-to-array rewriting

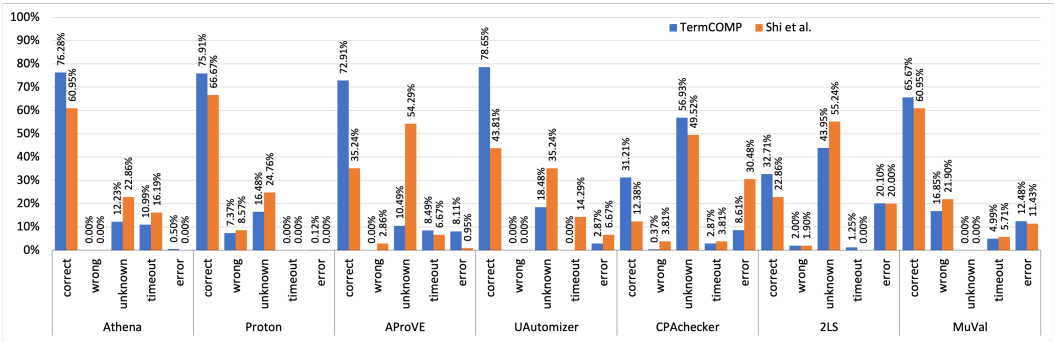


Fig. 7. Outcome distribution per tool on the TermCOMP and real-world Shi et al. benchmarks.

Table 1. Outcome breakdown per tool on the TermCOMP (TC) and real-world Shi et al. (RW) benchmarks. Each group of 4 columns shows the number of Wrong (W), Unknown (U), Timeout (T), and Error (E) results.

(a) Outcome breakdown per tool on the TermCOMP (TC) benchmarks.

Feature (#TC)	**Athena**				**Proton**				**AProVE**				**UAutomizer**				**CPAchecker**				**2LS**				**MuVal**			
	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E
Pointer Manipulation (143)	0	5	42	1	16	21	0	0	0	15	15	2	0	33	0	16	0	142	0	1	0	14	0	128	109	0	1	19
Data Structure (16)	0	5	10	0	13	0	0	0	0	1	0	0	0	15	0	0	0	16	0	0	0	14	0	1	8	0	0	7
Recursion (62)	0	38	0	0	0	23	0	0	0	11	2	4	0	17	0	0	0	7	0	55	9	3	0	9	11	0	0	0
Total (221)	0	48	52	1	29	44	0	0	0	27	17	6	0	65	0	16	0	165	0	56	9	31	0	138	128	0	1	26

(b) Outcome breakdown per tool on the real-world Shi et al. (RW) benchmarks.

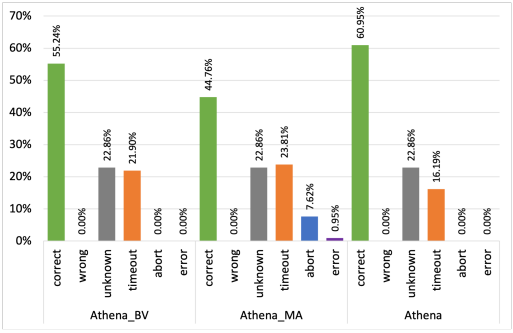
Feature (#RW)	**Athena**				**Proton**				**AProVE**				**UAutomizer**				**CPAchecker**				**2LS**				**MuVal**			
	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E	W	U	T	E
Pointer Manipulation (14)	0	0	11	0	3	5	0	0	0	13	1	0	0	1	9	1	0	8	0	6	0	12	0	2	4	0	0	6
Array (16)	0	0	10	0	1	7	0	0	0	13	1	0	0	3	6	0	0	14	0	2	0	11	0	2	3	0	2	4
Data Structure (6)	0	0	4	0	2	0	0	0	0	6	0	0	0	0	3	1	0	0	0	6	0	6	0	0	1	0	0	4
Bit Calculation (8)	0	0	0	0	0	1	0	0	0	8	0	0	0	4	0	0	0	8	0	0	0	6	0	0	4	0	0	0
Finite-Width Arithmetic Effects (23)	0	0	1	0	4	1	0	0	3	7	4	0	0	13	3	0	4	11	2	1	0	14	0	0	4	0	1	2
Recursion (24)	0	24	0	0	1	14	0	0	0	18	0	0	0	17	0	2	0	2	0	20	2	0	0	19	11	0	0	2
Total (91)	0	24	26	0	11	28	0	0	3	65	6	0	0	38	21	4	4	43	2	35	2	49	0	23	27	0	3	18

lacking sentinel-based reasoning, and (3) its use of quantified invariants via trace abstraction, which we leave for future work. Despite these omissions, matching UAutomizer closely while outperforming Proton and AProVE highlights Athena's strength. Reported percentages differ slightly from official TermCOMP 2024 and SV-COMP 2025 (Termination) results [51, 53] due to tool updates and system configuration. Overall, the results demonstrate Athena's reliability and correctness across synthetic and real-world benchmarks, addressing RQ1 and providing strong evidence for RQ3.

Although MuVal's correct rate appears competitive, a notable fraction of its results stems from incomplete semantic modeling. MuVal relies on pre-extension version of llvm2KITTeL (prior to this work) that does not translate key LLVM IR constructs—pointers, arrays, data structures, bitwise operations, bounded integer computations, and recursion—into LTS, instead silently omitting them and yielding models that do not faithfully represent the original programs. Our inspection showed MuVal reporting correct results despite missing translation support, including 14 TermCOMP programs with pointers, 1 with data structures, 51 with recursion, and 43 Shi et al. programs exercising these constructs. Such results may be formally sound under MuVal's abstract model but

Table 2. Average runtimes (in seconds) of each tool on the TermComp (TC) and Shi et al. (RW) benchmarks.

Metric	Athena		Proton		AProVE		UAutomizer		CPAchecker		2LS		MuVal	
	TC	RW	TC	RW	TC	RW	TC	RW	TC	RW	TC	RW	TC	RW
Avg. Runtime (s)	105.61	211.23	114.50	89.94	115.92	98.61	96.34	170.71	230.87	329.70	17.62	0.51	59.46	79.63



(a) Correctness distribution.

Feature (#RW)	Athena$_{BV}$					Athena$_{MA}$					Athena				
	W	U	T	A	E	W	U	T	A	E	W	U	T	A	E
Pointer Manipulation (14)	0	0	11	0	0	0	0	11	0	1	0	0	11	0	0
Array (16)	0	0	12	0	0	0	0	11	0	0	0	0	10	0	0
Data Structure (6)	0	0	4	0	0	0	0	3	0	1	0	0	4	0	0
Bit Calculation (8)	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0
Finite-Width Arithmetic Effects (23)	0	0	2	0	0	0	0	7	0	0	0	0	1	0	0
Recursion (24)	0	24	0	0	0	0	24	0	0	0	0	24	0	0	0
Total (91)	0	24	29	0	0	0	24	32	8	2	0	24	26	0	0

(b) Feature-level outcomes.

Metric	Athena$_{BV}$	Athena$_{MA}$	Athena
Avg. Runtime (s)	141.98	105.98	211.23

(c) Average runtimes.

Fig. 8. Comparative evaluation of Athena's semantic modes—Bit-Vector (Athena$_{BV}$) and Modulo Arithmetic (Athena$_{MA}$)—and their combination (Athena) on Shi et al. benchmarks.

are unreliable from the perspective of the original program semantics with respect to those omitted constructs. Similar issues may affect other tools and warrant further investigation. These findings underscore the value of Athena's semantic and memory-aware enhancements, as reflected in RQ4.

To address RQ2, Table 1 presents failure breakdowns across six C constructs—pointers, arrays, data structures, bitwise operations, finite-width arithmetic effects, and recursion—using separate tables for TermCOMP (TC) and Shi et al. (RW). For each construct, we report wrong (W), unknown (U), timeout (T), and error (E) outcomes. On TC, Athena fails on 48 of 143 pointer programs, competitive with AProVE (32), Proton (37), and UAutomizer (49), and far ahead of MuVal (129), 2LS (142), and CPAchecker (143). Despite llvm2KITTeL's function inlining preventing direct recursion analysis, Athena still solves 24 of 62 recursive programs by automatically converting tail recursion into loops. On RW, Athena records 50 failures, trailing Proton (39) but outperforming UAutomizer (63) and AProVE (74), highlighting its strength on complex real-world code. These achievements are particularly notable given that competing tools may report correctness under incomplete abstraction models, as discussed earlier for MuVal. Importantly, Athena maintains perfect soundness, producing zero wrong results across both benchmarks and all categories, matching UAutomizer and contrasting with all other tools, which report non-zero wrong results. Across the two bounded-arithmetic categories—finite-width arithmetic effects and bit calculation (31 programs total)—MuVal produces 8 wrong results, Proton and CPAchecker each 4, and AProVE 3, while Athena solves 30/31 correctly with zero wrong results. These results demonstrate the effectiveness of Athena's bounded-integer semantics assurance engine, expose the limitations of existing tools on bounded computations, and confirm Athena's ability to handle advanced C constructs—answering RQ2 and reinforcing RQ3.

Table 2 compares average runtimes across tools. On TermCOMP, Athena averages 105.61 s per program, close to UAutomizer (96.34 s), faster than Proton (114.50 s) and AProVE (115.92 s), and substantially faster than CPAchecker (230.87 s). On the real-world dataset, Athena averages 211.23 s, which is slower than Proton (89.94 s), AProVE (98.61 s), and UAutomizer (170.71 s), but still clearly faster than CPAchecker (329.70 s). In our evaluation, we observed that the higher runtime on real-world programs primarily stems from the complexity of Athena's termination/non-termination

analysis, rather than from pointer-to-array rewriting or bounded-integer semantics assurance alone. Overall, these results show that Athena remains practical and scales to real-world programs with acceptable runtime overhead, addressing RQ5.

Figure 8 addresses RQ6 by comparing Athena's two modes—bit-vector (Athena$_{BV}$) and modulo arithmetic (Athena$_{MA}$)—and their combination as Athena, across outcome distribution, feature-level breakdown, and average runtime. We distinguish between *abort* (A) and *error* (E). Athena$_{MA}$ incurs 7.62% aborted runs—all 8 benchmarks in the bit-calculation category—since these require bit-vector reasoning supported by Athena$_{BV}$. It also produces one out-of-memory error on a program involving pointers and data structures. Both aborts and this error are excluded from Athena's results under its combination strategy. As Athena$_{BV}$ successfully solves all 8 benchmarks aborted by Athena$_{MA}$, the two modes demonstrate similar overall capability. Their combination increases the correct rate from 55.24% (Athena$_{BV}$) and 44.76% (Athena$_{MA}$) to 60.95% (Athena), with each contributing solutions the other cannot. Athena$_{BV}$'s bitwidth-aware reasoning is most effective for bitwise operations and finite-width arithmetic effects, while Athena$_{MA}$'s modular arithmetic excels on pointers, arrays, and structures. Together, they expand coverage without compromising soundness, demonstrating the benefit of integration.

Among the evaluated (UB-free) benchmarks, we identified one misclassified program in the Shi et al. suite: `Incorrect_Initialization_2_T`, labeled as terminating but actually non-terminating. Athena, Proton, UAutomizer, and MuVal correctly report non-termination, while the remaining tools return unknown.

7 Conclusion and Future Work

This paper introduced Athena, a sound framework for termination and non-termination analysis of C programs under finite-width integer semantics, with support for pointers, arrays, structures, and bit-level arithmetic. Athena addresses key limitations of prior tools through several transformations and enhancements: pointer operations are rewritten as array accesses; bounded integer semantics are supported through modulo arithmetic and bit-vector semantics; and the resulting LTS enables modular reasoning via an extended version of MuVal. Our implementation builds on MuVal and llvm2KITTeL, both extended with new theories and abstractions. Experimental results show that Athena achieves high correctness on real-world benchmarks and delivers competitive performance on TermCOMP, while producing zero wrong results across both suites. These outcomes demonstrate its ability to reason soundly about complex memory patterns and low-level behaviors, thereby advancing the state of the art in C termination analysis.

While Athena demonstrates strong capabilities in analyzing complex C programs, several directions remain for enhancement. First, we plan to augment the pointer-to-array rewriting phase with reasoning about sentinel-based idioms, leveraging array size information to model patterns such as null-terminated strings. We also aim to add memory safety checks (out-of-bounds and null dereference) and support for `free`, enabling Athena to reason about unsafe accesses and dynamic memory lifecycles. Second, we intend to enhance MuVal with quantified invariant synthesis, complementing its current CEGIS-based approach and enabling Athena to handle benchmarks that require universally quantified reasoning while preserving its ability to solve problems beyond the reach of simpler invariant generation methods. Third, we will extend Athena to directly support recursive procedures. Our pipeline relies on aggressive inlining via llvm2KITTeL, which simplifies control flow but prevents recursion. We aim to design an abstraction that preserves call hierarchies without inlining, allowing sound reasoning about recursive programs. Finally, we plan to extend Athena to handle complex types such as C unions and C++ classes through specialized abstractions that accurately capture their semantics and memory layouts. These extensions will further broaden Athena's applicability to safety-critical and industrial-scale systems.

Acknowledgments

We thank the anonymous reviewers for their insightful comments that helped improve the quality of this paper. We would also like to thank Kazuki Uehara and Takuma Monma for their contributions to the implementation of Athena. This study was partially supported by JSPS KAKENHI Grant Numbers JP25H00446 and JP25K24739.

8 Data Availability

The Athena implementation is publicly available at: https://github.com/negarfathi/Athena, which includes installation instructions, benchmarks, execution scripts, and evaluation results needed to reproduce the experiments in this paper. We also refer readers to the maintained evaluation infrastructure at: https://github.com/hiroshi-unno/coar for future reuse and comparative evaluation. To support long-term preservation and citation, the artifact is also archived on Zenodo [23].

References

[1] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Proceedings of the 17th International Symposium on Static Analysis* (Perpignan, France) *(SAS 2010)*. Springer, Berlin, Heidelberg, 117–133. doi:10.1007/978-3-642-15769-1_8

[2] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. 2012. Detecting Fair Non-termination in Multithreaded Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification* (Berkeley, CA, USA) *(CAV 2012)*. Springer, Berlin, Heidelberg, 210–226. doi:10.1007/978-3-642-31424-7_19

[3] Alexey Bakhirkin, Josh Berdine, and Nir Piterman. 2015. A Forward Analysis for Recurrent Sets. In *Proceedings of the 22nd International Symposium on Static Analysis* (Saint-Malo, France) *(SAS 2015)*. Springer, Berlin, Heidelberg, 293–311. doi:10.1007/978-3-662-48288-9_17

[4] Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4, Article 26 (July 2014), 55 pages. doi:10.1145/2629488

[5] Amir M. Ben-Amram and Samir Genaim. 2017. On Multiphase-Linear Ranking Functions. In *Proceedings of the 29th International Conference on Computer Aided Verification* (Heidelberg, Germany) *(CAV 2017)*. Springer, Cham, 601–620. doi:10.1007/978-3-319-63390-9_32

[6] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT, USA) *(CAV 2011)*. Springer, Berlin, Heidelberg, 184–190. doi:10.1007/978-3-642-22110-1_16

[7] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Proceedings of the 17th International Conference on Computer Aided Verification* (Edinburgh, Scotland, UK) *(CAV 2005)*. Springer, Berlin, Heidelberg, 491–504. doi:10.1007/11513988_48

[8] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. The Polyranking Principle. In *Proceedings of the 32nd International Conference on Automata, Languages and Programming* (Lisbon, Portugal) *(ICALP 2005)*. Springer, Berlin, Heidelberg, 1349–1361. doi:10.1007/11523468_109

[9] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Termination Analysis of Integer Linear Loops. In *Proceedings of the 16th International Conference on Concurrency Theory* (San Francisco, CA, USA) *(CONCUR 2005)*. Springer, Berlin, Heidelberg, 488–502. doi:10.1007/11539452_37

[10] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Eindhoven, The Netherlands) *(TACAS 2016)*. Springer, Berlin, Heidelberg, 387–393. doi:10.1007/978-3-662-49674-9_22

[11] Marek Chalupa. 2024. *mchalupa/dg*. Retrieved April 15, 2026 from https://github.com/mchalupa/dg

[12] CHC-COMP Committee. 2024. *CHC-COMP: The Constrained Horn Clause Verification Competition*. Retrieved April 15, 2026 from https://chc-comp.github.io/

[13] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Proving Nontermination via Safety. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Grenoble, France) *(TACAS 2014)*. Springer, Berlin, Heidelberg, 156–171. doi:10.1007/978-3-642-54862-8_11

[14] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2015. Synthesising Interprocedural Bit-Precise Termination Proofs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska, USA) *(ASE '15)*. IEEE Press, Piscataway, NJ, USA, 53–64. doi:10.1109/ASE.2015.10

[15] Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2018. Bit-Precise Procedure-Modular Termination Analysis. *ACM Trans. Program. Lang. Syst.* 40, 1, Article 1 (March 2018), 38 pages. doi:10.1145/3121136

[16] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Genova, Italy) *(TACAS 2001)*. Springer, Berlin, Heidelberg, 67–81. doi:10.1007/3-540-45319-9_6

[17] Michael Colón and Henny Sipma. 2002. Practical Methods for Proving Program Termination. In *Proceedings of the 14th International Conference on Computer Aided Verification* (Copenhagen, Denmark) *(CAV 2002)*. Springer, Berlin, Heidelberg, 442–454. doi:10.1007/3-540-45657-0_36

[18] Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Disproving Termination with Overapproximation. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design* (Lausanne, Switzerland) *(FMCAD '14)*. FMCAD Inc, Austin, Texas, 67–74. doi:10.1109/FMCAD.2014.6987597

[19] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 415–426. doi:10.1145/1133981.1134029

[20] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving Program Termination. *Commun. ACM* 54, 5 (May 2011), 88–98. doi:10.1145/1941487.1941509

[21] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) *(TACAS 2013)*. Springer, Berlin, Heidelberg, 47–61. doi:10.1007/978-3-642-36742-7_4

[22] Stephan Falke, Deepak Kapur, and Carsten Sinz. 2011. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications* (Novi Sad, Serbia) *(RTA 2011)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 41–50. doi:10.4230/LIPIcs.RTA.2011.41

[23] Negar Fathi, Hiroshi Unno, Tachio Terauchi, and Rahul Purandare. 2026. *Athena Artifact.* Zenodo, Geneva, Switzerland. doi:10.5281/zenodo.19455305

[24] FSE2022benchmarks. 2022. *GitHub - FSE2022benchmarks/-FSE-2022-Termination at v1.0.* Retrieved April 15, 2026 from https://github.com/FSE2022benchmarks/-FSE-2022-Termination/tree/v1.0

[25] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2014. Proving Termination of Programs Automatically with AProVE. In *Proceedings of the 7th International Joint Conference on Automated Reasoning* (Vienna, Austria) *(IJCAR 2014)*. Springer, Cham, 184–191. doi:10.1007/978-3-319-08587-6_13

[26] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-termination. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 147–158. doi:10.1145/1328438.1328459

[27] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Proceedings of the 17th International Symposium on Static Analysis* (Perpignan, France) *(SAS 2010)*. Springer, Berlin, Heidelberg, 304–319. doi:10.1007/978-3-642-15769-1_19

[28] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. 2018. Ultimate Automizer and the Search for Perfect Interpolants (Competition Contribution). In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Thessaloniki, Greece) *(TACAS 2018)*. Springer, Cham, 447–451. doi:10.1007/978-3-319-89963-3_30

[29] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate Automizer with SMTInterpol (Competition Contribution). In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) *(TACAS 2013)*. Springer, Berlin, Heidelberg, 641–643. doi:10.1007/978-3-642-36742-7_53

[30] Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. 2016. Ultimate Automizer with Two-track Proofs (Competition Contribution). In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Eindhoven, The Netherlands) *(TACAS 2016)*. Springer, Berlin, Heidelberg, 950–953. doi:10.1007/978-3-662-49674-9_68

[31] Matthias Heizmann, Daniel Dietsch, Jan Leike, Betim Musa, and Andreas Podelski. 2015. Ultimate Automizer with Array Interpolation (Competition Contribution). In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (London, UK) *(TACAS 2015)*. Springer, Berlin, Heidelberg, 455–457. doi:10.1007/978-3-662-46681-0_43

[32] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification* (Vienna, Austria) *(CAV*

2014). Springer, Cham, 797–813. doi:10.1007/978-3-319-08867-9_53

[33] Jera Hensel, Frank Emrich, Florian Frohn, Thomas Ströder, and Jürgen Giesl. 2017. AProVE: Proving and Disproving Termination of Memory-Manipulating C Programs (Competition Contribution). In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Uppsala, Sweden) *(TACAS 2017)*. Springer, Berlin, Heidelberg, 350–354. doi:10.1007/978-3-662-54580-5_21

[34] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. 2016. Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution. In *Proceedings of the 14th International Conference on Software Engineering and Formal Methods* (Vienna, Austria) *(SEFM 2016)*. Springer, Cham, 234–252. doi:10.1007/978-3-319-41591-8_16

[35] Hrishikesh Karmarkar, Raveendra Kumar Medicherla, Ravindra Metta, and Prasanth Yeduru. 2022. FuzzNT: Checking for Program Non-termination. In *Proceedings of the 2022 IEEE International Conference on Software Maintenance and Evolution* (Limassol, Cyprus) *(ICSME 2022)*. IEEE Computer Society, Los Alamitos, CA, USA, 409–413. doi:10.1109/ICSME55016.2022.00049

[36] Anvesh Komuravelli, Nikolaj S. Bjørner, Arie Gurfinkel, and Kenneth L. McMillan. 2015. Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design* (Austin, Texas, USA) *(FMCAD 2015)*. IEEE, Piscataway, NJ, USA, 89–96. doi:10.1109/FMCAD.2015.7542257

[37] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV 2010)*. Springer, Berlin, Heidelberg, 89–103. doi:10.1007/978-3-642-14295-6_9

[38] Satoshi Kura, Hiroshi Unno, and Ichiro Hasuo. 2021. Decision Tree Learning in CEGIS-Based Termination Analysis. In *Proceedings of the 33rd International Conference on Computer Aided Verification* (Virtual Event) *(CAV 2021)*. Springer, Cham, 75–98. doi:10.1007/978-3-030-81688-9_4

[39] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving Non-termination Using Max-SMT. In *Proceedings of the 26th International Conference on Computer Aided Verification* (Vienna, Austria) *(CAV 2014)*. Springer, Cham, 779–796. doi:10.1007/978-3-319-08867-9_52

[40] Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 189 (Nov. 2020), 30 pages. doi:10.1145/3428257

[41] Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Grenoble, France) *(TACAS 2014)*. Springer, Berlin, Heidelberg, 172–186. doi:10.1007/978-3-642-54862-8_12

[42] Viktor Malík, Štefan Martiček, Peter Schrammel, Mandayam Srivas, Tomáš Vojnar, and Johanan Wahlang. 2018. 2LS: Memory Safety and Non-termination. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Thessaloniki, Greece) *(TACAS 2018)*. Springer, Cham, 417–421. doi:10.1007/978-3-319-89963-3_24

[43] Ravindra Metta, Hrishikesh Karmarkar, Kumar Madhukar, R. Venkatesh, and Supratik Chakraborty. 2024. PROTON: PRObes for Termination Or Not (Competition Contribution). In *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Luxembourg City, Luxembourg) *(TACAS 2024)*. Springer, Cham, 393–398. doi:10.1007/978-3-031-57256-2_27

[44] Diganta Mukhopadhyay, Ravindra Metta, Hrishikesh Karmarkar, and Kumar Madhukar. 2025. PROTON 2.1: Synthesizing Ranking Functions via fine-tuned locally Hosted LLM (Competition Contribution). In *Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Hamilton, ON, Canada) *(TACAS 2025)*. Springer, Cham, 242–247. doi:10.1007/978-3-031-90660-2_19

[45] Eike Neumann, Joël Ouaknine, and James Worrell. 2020. On Ranking Function Synthesis and Termination for Polynomial Programs. In *Proceedings of the 31st International Conference on Concurrency Theory* (Vienna, Austria (Virtual Conference)) *(CONCUR 2020)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:15. doi:10.4230/LIPIcs.CONCUR.2020.15

[46] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation* (Venice, Italy) *(VMCAI 2004)*. Springer, Berlin, Heidelberg, 239–251. doi:10.1007/978-3-540-24622-0_20

[47] Azalea Raad, Julien Vanegue, and Peter O'Hearn. 2024. Non-termination Proving at Scale. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 280 (Oct. 2024), 29 pages. doi:10.1145/3689720

[48] Peter Schrammel and Daniel Kroening. 2016. 2LS for Program Analysis (Competition Contribution). In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Eindhoven, The Netherlands) *(TACAS 2016)*. Springer, Berlin, Heidelberg, 905–907. doi:10.1007/978-3-662-49674-9_56

[49] Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. 2022. Large-scale Analysis of Non-termination Bugs in Real-world OSS Projects. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering (Singapore, Singapore) *(ESEC/FSE 2022)*. ACM, New York, NY, USA, 256–268. doi:10.1145/3540250.3549129

[50] Thomas Ströder, Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jürgen Giesl. 2015. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (London, United Kingdom) *(TACAS 2015)*. Springer, Berlin, Heidelberg, 417–419. doi:10.1007/978-3-662-46681-0_32

[51] SV-COMP Organizers. 2025. *SV-COMP 2025: Termination Verified Results*. Retrieved April 15, 2026 from https://sv-comp.sosy-lab.org/2025/results/results-verified/META_Termination.table.html#/

[52] TermCOMP Contributors. 2024. *TermCOMP/TPDB*. Retrieved April 15, 2026 from https://github.com/TermCOMP/TPDB

[53] Termination Competition Organizers. 2024. *Termination Competition 2024*. Retrieved April 15, 2026 from https://termcomp.github.io/Y2024/

[54] The LLVM Project. 2019. *The LLVM Compiler Infrastructure Project*. Retrieved April 15, 2026 from https://llvm.org/

[55] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. *Proc. ACM Program. Lang.* 7, POPL, Article 72 (Jan. 2023), 30 pages. doi:10.1145/3571265

[56] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *Proceedings of the 33rd International Conference on Computer Aided Verification* (Virtual Event) *(CAV 2021)*. Springer, Cham, 742–766. doi:10.1007/978-3-030-81685-8_35

[57] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *Proceedings of the 20th International Symposium on Static Analysis* (Seattle, WA, USA) *(SAS 2013)*. Springer, Berlin, Heidelberg, 43–62. doi:10.1007/978-3-642-38856-9_5

[58] Helga Velroyen and Philipp Rümmer. 2008. Non-termination Checking for Imperative Programs. In *Proceedings of the 2nd International Conference on Tests and Proofs* (Prato, Italy) *(TAP 2008)*. Springer, Berlin, Heidelberg, 154–170. doi:10.1007/978-3-540-79124-9_11

[59] Wei Wang, Clark Barrett, and Thomas Wies. 2017. Partitioned Memory Models for Program Analysis. In *Proceedings of the 18th International Conference on Verification, Model Checking, and Abstract Interpretation* (Paris, France) *(VMCAI 2017)*. Springer, Cham, 539–558. doi:10.1007/978-3-319-52234-0_29

[60] Xiaofei Xie, Bihuan Chen, Liang Zou, Shang-Wei Lin, Yang Liu, and Xiaohong Li. 2017. Loopster: static loop termination analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 84–94. doi:10.1145/3106237.3106260

[61] Yao Zhang, Xiaofei Xie, Yi Li, Sen Chen, Cen Zhang, and Xiaohong Li. 2023. EndWatch: A Practical Method for Detecting Non-Termination in Real-World Software. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE, Piscataway, NJ, USA, 686–697. doi:10.1109/ASE56229.2023.00061

Received 2025-09-11; accepted 2026-03-24