Exploring Locally Bounded Translation Validation

Alborz Jelvani
Rutgers University
Department of Computer Science
Piscataway, USA
alborz.jelvani@rutgers.edu

Richard P. Martin
Rutgers University
Department of Computer Science
Piscataway, USA
rmartin@scarletmail.rutgers.edu

Santosh Nagarakatte
Rutgers University
Department of Computer Science
Piscataway, USA
santosh.nagarakatte@cs.rutgers.edu

Abstract

Bounded translation validation (TV) is a common method used to check for the equivalence of two programs by unrolling all loops according to a user-supplied global bound. Unfortunately, large unroll bounds with multiple loops can increase the state space and the complexity of the TV. This paper proposes a systematic strategy for applying *local* unroll bounds for bounded TV of program pairs with loops. Our technique increases the coverage of the bounded TV in cases where the TV would otherwise timeout with the existing global unroll bounds. We introduce a lattice-based formulation of locally bounded program pairs and demonstrate that random sampling of local unroll bounds from this lattice admits a search-and-prune strategy that is effective for verification.

We have modified the Alive2 bounded TV to support local unroll bounds and evaluated our approach on benchmarks such as TSVC. The results show an increase in verification coverage with our lattice search-and-prune strategy combined with local unroll bounds.

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

Keywords: Translation Validation, Compilers

ACM Reference Format:
Alborz Jelvani, Richard P. Martin, and Santosh Nagarakatte. 2026. Exploring Locally Bounded Translation Validation. In *Proceedings of the 15th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '26), June 15–19, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3814987.3814994

1 Introduction

Equivalence checking between two programs is an important problem with applications in compiler verification [11–13, 15, 21, 22, 24], the verification of LLM-generated code [27], and program synthesis [19]. Automatic equivalence checking of two arbitrary programs is undecidable. Hence, various heuristics are used to check equivalence in the presence of unbounded loops [4, 8, 25].

Although determining whether two programs are equivalent is undecidable, approximations are still very useful. Bounded translation validation is an approach that under-approximates equivalence, meaning it can prove the presence of bugs but not necessarily their absence. A bounded TV unrolls loops (and recursive procedures) up to a user-specified bound, reducing the programs to straight-line code and branches. These programs are then encoded with Satisfiability Modulo Theory (SMT) solvers using the quantifier-free theory of bit-vectors and arrays. The equivalence problem then becomes decidable, but only up to the given unroll bound.

Exploring the equivalence of programs using bounded verification has many important uses. For example, when verifying code refactored by LLMs the larger the unroll bound the higher the confidence we can have in the equivalence. Compiler correctness can also be aided by using bounded-checking; any counter-example found between the original and optimized code usually means there is a bug in the compiler.

To automatically check for equivalence, SMT solvers are often employed. However, even with small loop unroll bounds (less than five) the state-space of the logic formulas can increase to beyond what is computationally feasible. Thus, the loop unroll bounds control the trade-off between precision and tractability of a bounded TV.

In this paper, we propose a search-and-prune strategy to control the bounds of individual program loops to maximize the overall coverage of a bounded translation validator. We show how to characterize the set of local loop bounds as a lattice where some sets of loop bounds dominate others; that is one set of loop bounds proves equivalence for another dominated set of loop bounds. We can then explore the lattice in a manner to maximize program coverage, enabling coverage not possible using simpler strategies, such as giving all loops the same bound.

To demonstrate our approach, we modify Alive2 which is a popular bounded-translation validator for LLVM IR programs [15]. Originally started as an effort to check the correctness of peephole optimizations [13, 14], it has evolved to support bounded translation validation. It has detected hundreds of compiler optimization bugs in LLVM. When given a pair of programs (source, target) with loops to the Alive2 TV

SOAP '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2709-2/2026/06
https://doi.org/10.1145/3814987.3814994

```
1 void func(int* a, int* b, int* c, int* d, int
      iterations, int LEN_1D) {
2     for (int nl = 0; nl < iterations; nl++) {
3         for (int i = 0; i < LEN_1D; i++) {
4   +           if(i < 2) {
5                   a[i] += b[i] * c[i];
6   +           }
7 }     }     }
```

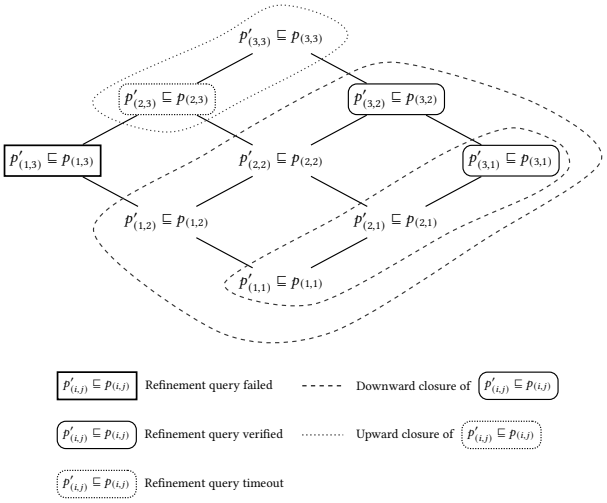

Figure 1. Top: Example of a program before and after the addition of the green highlights. The bounded TV will conclude these two programs are equivalent unless a sufficiently large unroll bound is used. **Bottom:** The lattice searched for the program pairs when a global upper bound of $k = 3$ is given to our tool. The query for node $p'_{(1,3)} \sqsubseteq p_{(1,3)}$ revealed the bug. The solid rounded rectangular nodes represent refinement queries verified by the TV, with the enclosed in dashed regions representing their downward closures. The dotted rounded rectangular node represents TV time out queries with the enclosed dotted region representing their upward closure.

tool (`alive-tv`), it unrolls each loop a fixed number of times in the source and the target program. This unroll bound can be specified by the user but is globally applied to all loops. The loops are unrolled inside-out, with the inner most loops of each loop nest being unrolled before the outer loops. The `alive-tv` tool checks the resulting unrolled source and target programs for refinement. Although `alive-tv` may report a program pair under a given unroll bound is a refinement, a counter-example may exist if a larger unroll bound is used.

Bounded TV Limitations. Consider the code listing in Figure 1. This function contains a pair of nested loops, where the code highlighted in green is a proposed edit to the code. Our goal is to check whether the addition of the highlighted green code preserves equivalence between the original program for all inputs to this function. When checking with the `alive-tv` tool if this code (compiled to LLVM IR) with and without the highlighted lines of code is equivalent, `alive-tv` reports that the pair of programs seem to be equivalent. If we apply a loop unroll factor below 3, `alive-tv` will not be able to consider the case of $i = 2$, which would skip the summation on line 5 introducing nonequivalent behavior.

Using large unroll bounds globally increases the size of the resulting program after the loops are unrolled, which makes the verification of resulting programs intractable. The `alive-tv` tool may time out in such scenarios.

To uncover bugs similar to the one in Figure 1, we propose *local unroll bounds*, which are unroll bounds applied to individual loops in the program. In Figure 1, the number of iterations in the inner and outer loops is completely independent of each other. Therefore, one possible behavior of this program may involve the inner loop iterating three times while the outer loop iterates only once. If we apply such unroll bounds locally, the bounded program will only consist of three loop bodies as opposed to the nine loop bodies created with a global unroll bound of three, and indeed this becomes a much more tractable verification task for a bounded TV.

Systematic Exploration of Local Unroll Bounds with a Partial Order Directed Traversal. To make the exploration of such local unroll bounds systematic, we propose a framework to search the space of locally unrolled programs that maximizes coverage for a given time budget. We define a lattice that represents all possible locally unrolled programs up to a global bound and a partial ordering between the unrolled programs. Given two (correct) translation validation instances $p'_i \sqsubseteq p_i$ and $p'_{i'} \sqsubseteq p_{i'}$ where the programs p' and p have loops globally unrolled i and i' times respectively, we say $(p'_i \sqsubseteq p_i) \preceq (p'_{i'} \sqsubseteq p_{i'})$ if $i \leq i'$. Intuitively, a program pair verified with the larger global unrolling i' implies the verification of the program pair verified with the smaller global unrolling factor of i. This follows because the larger unrolled pair of programs contain all behaviors of the smaller unrolled program pairs. This partial order applied to the space of locally unrolled programs can be iteratively traversed and dispatched to the translation validator, pruning respective lattice nodes on timeout or verified queries.

We have modified Alive2 to support local unroll bounds for loops. The tool `alive-tv` now accepts a list of unroll bounds for loops in the `src` and `tgt` functions separately as opposed to the single global unroll bound, which was previously the default. This addition alone allows for better tractability in detecting deep bugs, similar to the one in Figure 1. When applied correctly, local unroll bounds can prevent timeouts caused by explosions in code size and allow the user to explore loop nesting relations to attain richer coverage.

Our modifications to the `alive-tv` tool are publicly available [9]. We have used our modified `alive-tv` tool to demonstrate increased verification coverage of programs containing loops that time out, such as those from the TSVC benchmark suite [17].

2 Loops with Local Unroll Bounds

In this section, we define local loop unrollings applied to translation validation instances, as well as an ordering relation between these instances.

2.1 The Lattice of Locally Unrolled Programs

Let p denote an arbitrary program, with an optimized version denoted as p'. We use a subscript such as k on a program p, denoted as p_k, to represent a global unrolling bound for all loops in program p. This means that each loop is unrolled k times. Additionally, we use a tuple as a subscript to denote a program where loops are locally unrolled according to the tuple's values, in the order they appear in the code. For example, $p_{(1,2,3)}$ is the program p where the last loop is unrolled three times, the second loop is unrolled two times, and the first loop is unrolled once (unrolling is performed inside-out).

Defining Program Refinement. Let $P' \times P$ represent the set of all pairs of program unrolling assignments for programs P' and P and their loops respectively. For simplicity, we assume all program pairs have the same number of loops. The refinement relation $\sqsubseteq$ is defined for elements of $P' \times P$, where, given $p' \in P'$ and $p \in P$, we let $p' \sqsubseteq p$ denote that program p' refines p. This refinement relation is checked by the TV for validity. If such a refinement relation does not hold (i.e. there is a bug in one program) the TV will reject the pair. A third possibility is a TV timeout, in which case the status of the refinement relation between the program pair remains unknown.

Defining a Refinement Lattice. Let $P^{\sqsubseteq}_{k^n}$ denote a set of unrolled program pairs from $P' \times P$ each with n loops verified up to global bound k and related with the refinement relation $\sqsubseteq$. Let $\preceq$ denote a partial ordering between these refinement relations, contained in the set $(P^{\sqsubseteq}_{k^n}, \preceq)$.

For refinement relations $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$ and $p'_j \sqsubseteq p_j \in P^{\sqsubseteq}_{k^n}$, the ordering relation $(p'_i \sqsubseteq p_i) \preceq (p'_j \sqsubseteq p_j) \in (P^{\sqsubseteq}_{k^n}, \preceq)$ if an only if $i \leq j$. Note that if i and j are tuples, the ordering $i \leq j$ is defined such that each element of i is less than or equal to the corresponding element in j.

Intuitively, this means if a refinement relation holds for a loop unrolling assignment j, it must also hold for those same program pairs under any smaller unrolling factor $i \leq j$. In other words, the refinement relation for the programs with the larger unroll bound is more precise than the refinement relation for the programs with the smaller unroll bound.

To illustrate an example of local unrolling refinement relations for a program, consider the program pairs given in Figure 1. This program has two loops and the refinement relations were verified up to an unroll bound of three as given in Figure 1. We can define the set of refined program pairs as $P^{\sqsubseteq}_{(3,3)} = \{p'_{(1,1)} \sqsubseteq p_{(1,1)}, p'_{(1,2)} \sqsubseteq p_{(1,2)}, p'_{(2,1)} \sqsubseteq p_{(2,1)}, p'_{(2,2)} \sqsubseteq p_{(2,2)}, p'_{(3,1)} \sqsubseteq p_{(3,1)}, p'_{(3,2)} \sqsubseteq p_{(3,2)}\}$. Finally, the ordering relation $\preceq$ can be applied to pairs from $P^{\sqsubseteq}_{(3,3)}$ whenever a tuple subscript can be ordered by $\leq$ as previously defined. The ordering relation $\preceq$ applied to $P^{\sqsubseteq}_{(3,3)}$ does not form a totally ordered set because element pairs such as $(p'_{(1,2)} \sqsubseteq p_{(1,2)}, p'_{(2,1)} \sqsubseteq p_{(2,1)})$ cannot be ordered using our definition of $\leq$ on tuples. Intuitively, this restriction means the program $p_{(1,2)}$ is not more or less precise than $p_{(2,1)}$, and no ordering relation can be applied. Notice that $|P^{\sqsubseteq}_{k^n}| = k^n$.

We now formally define the partial order $\preceq$ on $P^{\sqsubseteq}_{k^n}$.

Theorem 2.1. *$(P^{\sqsubseteq}_{k^n}, \preceq)$ forms a partially ordered set and satisfies the three properties below for any $p'_i \sqsubseteq p_i, p'_j \sqsubseteq p_j, p'_l \sqsubseteq p_l \in P^{\sqsubseteq}_{k^n}$:*

1. ***Reflexive**: $p'_i \sqsubseteq p_i \preceq p'_i \sqsubseteq p_i$*
2. ***Anti-symmetric**: $p'_i \sqsubseteq p_i \preceq p'_j \sqsubseteq p_j \wedge p'_j \sqsubseteq p_j \preceq p'_i \sqsubseteq p_i \implies p'_j \sqsubseteq p_j = p'_i \sqsubseteq p_i$*
3. ***Transitive**: $p'_i \sqsubseteq p_i \preceq p'_j \sqsubseteq p_j \wedge p'_j \sqsubseteq p_j \preceq p'_l \sqsubseteq p_l \implies p'_i \sqsubseteq p_i \preceq p'_l \sqsubseteq p_l$*

Additionally, we define the lattice formed by $(P^{\sqsubseteq}_{k^n}, \preceq)$.

Theorem 2.2. *The partially ordered set $(P^{\sqsubseteq}_{k^n}, \preceq)$ forms a bounded lattice. For any elements $p'_i \sqsubseteq p_i, p'_j \sqsubseteq p_j \in P^{\sqsubseteq}_{k^n}$, the following properties hold:*

1. ***Join**: Every pair $p'_i \sqsubseteq p_i, p'_j \sqsubseteq p_j$ has a unique supremum $p'_i \sqsubseteq p_i \vee p'_j \sqsubseteq p_j \in P^{\sqsubseteq}_{k^n}$ such that $p'_i \sqsubseteq p_i \preceq p'_i \sqsubseteq p_i \vee p'_j \sqsubseteq p_j$ and $p'_j \sqsubseteq p_j \preceq p'_i \sqsubseteq p_i \vee p'_j \sqsubseteq p_j$. Furthermore, for any $p'_l \sqsubseteq p_l \in P^{\sqsubseteq}_{k^n}$ where $p'_i \sqsubseteq p_i \preceq p'_l \sqsubseteq p_l$ and $p'_j \sqsubseteq p_j \preceq p'_l \sqsubseteq p_l$, it holds that $p'_i \sqsubseteq p_i \vee p'_j \sqsubseteq p_j \preceq p'_l \sqsubseteq p_l$.*
2. ***Meet**: Every pair $p'_i \sqsubseteq p_i, p'_j \sqsubseteq p_j$ has a unique infimum $p'_i \sqsubseteq p_i \wedge p'_j \sqsubseteq p_j \in P^{\sqsubseteq}_{k^n}$ such that $p'_i \sqsubseteq p_i \wedge p'_j \sqsubseteq p_j \preceq p'_i \sqsubseteq p_i$ and $p'_i \sqsubseteq p_i \wedge p'_j \sqsubseteq p_j \preceq p'_j \sqsubseteq p_j$. Furthermore, for any $p'_l \sqsubseteq p_l \in P^{\sqsubseteq}_{k^n}$ where $p'_l \sqsubseteq p_l \preceq p'_i \sqsubseteq p_i$ and $p'_l \sqsubseteq p_l \preceq p'_j \sqsubseteq p_j$, it holds that $p'_l \sqsubseteq p_l \preceq p'_i \sqsubseteq p_i \wedge p'_j \sqsubseteq p_j$.*
3. ***Top**: There exists a unique element $\top \in P^{\sqsubseteq}_{k^n}$ such that for all $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$, $p'_i \sqsubseteq p_i \preceq \top$.*
4. ***Bottom**: There exists a unique element $\bot \in P^{\sqsubseteq}_{k^n}$ such that for all $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$, $\bot \preceq p'_i \sqsubseteq p_i$.*

We omit the proof for these theorems as they directly follow from the lattice formed by the cartesian products of the lattice $(\mathbb{Z}, \leq)$.

Now we explain these definitions in the context of translation validation. $\top \in P^{\sqsubseteq}_{k^n}$ represents the refinement relation $p_{k^n} \sqsubseteq p_{k^n}$. If this refinement relation holds, then the refinement relation for all possible remaining $k^n - 1$ elements in $P^{\sqsubseteq}_{k^n}$ also holds. Conversely, $\bot \in (P^{\sqsubseteq}_{k^n}, \preceq)$ represents the refinement relation $p_{1^n} \sqsubseteq p_{1^n}$. If this refinement relation *does not* hold, it follows that that all other refinement relations in $P^{\sqsubseteq}_{k^n}$ do not hold.

In order to represent how verifying some element of $P^{\sqsubseteq}_{k^n}$ impacts the lattice $(P^{\sqsubseteq}_{k^n}, \preceq)$ we define the following.

Definition 1. For any element $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$, the *lower closure* of $p'_i \sqsubseteq p_i$ in $(P^{\sqsubseteq}_{k^n}, \preceq)$, denoted by $\downarrow p'_i \sqsubseteq p_i$, is defined as the set:

$$\downarrow p'_i \sqsubseteq p_i = \{p'_j \sqsubseteq p_j \in P^{\sqsubseteq}_{k^n} \mid p'_j \sqsubseteq p_j \preceq p'_i \sqsubseteq p_i\}.$$

Similarly, the *upper closure* of $p'_i \sqsubseteq p_i$ in $(P^{\sqsubseteq}_{k^n}, \preceq)$, denoted by $\uparrow p'_i \sqsubseteq p_i$, is defined as:

$$\uparrow p'_i \sqsubseteq p_i = \{p'_j \sqsubseteq p_j \in P^{\sqsubseteq}_{k^n} \mid p'_i \sqsubseteq p_i \preceq p'_j \sqsubseteq p_j\}.$$

2.2 Searching the Lattice

This section describes how these definitions are used to build a search strategy for exploring local unroll bounds for translation validation. Since we randomly sample elements of the lattice to verify, we may have k^n queries to the TV where k represents the upper bound for unrolling and n represents the number of loops in our program pairs. But in the best case we may verify $\top$ which will imply all other refinement relations in the lattice hold because $\downarrow \top = P^{\sqsubseteq}_{k^n}$.

The high-level idea of our search approach is to randomly select an element from $P^{\sqsubseteq}_{k^n}$ and query the TV. Then we handle the three cases below:

1. **Correct**: If $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$ terminates and verifies to be correct we mark all elements in $\downarrow p'_i \sqsubseteq p_i$ as verified.
2. **Timeout**: If $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$ times out we mark all elements in $\uparrow p'_i \sqsubseteq p_i$ as timeouts.
3. **Incorrect**: If $p'_i \sqsubseteq p_i \in P^{\sqsubseteq}_{k^n}$ terminates and fails the refinement check then we terminate and return with the counter-example.

Each time an element from $P^{\sqsubseteq}_{k^n}$ is sampled and these three checks are performed, we then pick a new unmarked element from $P^{\sqsubseteq}_{k^n}$ and continue this process until all elements of $P^{\sqsubseteq}_{k^n}$ have been marked. Our search-and-prune strategy is provided in Algorithm 1. Our algorithm still takes a user-supplied global unroll bound k, but unlike using a static global unroll bound which may cause a timeout, our approach allows us incrementally attempt verification queries through the space of local unroll bounds without exhaustively enumerating all possible combinations.

Algorithm 1 Lattice search-and-prune procedure.

Require: Set of all lattice nodes $P^{\sqsubseteq}_{k^n}$, the translation validator TV, and an SMT solver SOLVER.
Ensure: Every node in U is eventually added to $V_\downarrow \cup T_\uparrow$.

```
U ← P^⊑_{k^n}                          ▷ Set of unmarked nodes
V ← ∅, T ← ∅                           ▷ Set of verified and timeout nodes
V_↓ ← ∅, T_↑ ← ∅                       ▷ Lower and upper closure of nodes
while U ≠ ∅ do
    node ← SOLVER(U)                   ▷ Randomly sample a node
    result ← TV(node)                  ▷ Check candidate refinement
    if result is verified then
        V ← V ∪ node
        V_↓ ← V_↓ ∪ ↓ node
        U ← U \ ↓ node
    else if result is timeout then
        T ← T ∪ {node}
        T_↑ ← T_↑ ∪ ↑ node
        U ← U \ ↑ node
    else if result is counter-example then
        return result
    end if
end while
return V, T
```

3 Modifications to `alive-tv`

We modify `alive-tv` to support local unroll bounds. The modified tool can be found at https://github.com/jelvani/Alive2-custom. The tool now takes a comma-separated list of integers as local unroll bounds for the `src-unroll` and `tgt-unroll` arguments of `alive-tv`. These unroll bounds are applied to loops in the order they appear in the code.

Our modifications to `alive-tv` preserve both *soundness* and *compatibility*. To preserve soundness, our tool makes no modifications to the main unroll logic used by `alive-tv`. Specifically, we ensure that for each loop `alive-tv` still inserts assertions ensuring the loop does not exceed the corresponding unroll factor that is applied. This introduces a constraint to the SMT solver, asserting that program paths containing loop induction variable assignments obtained beyond the provided unroll bound are not considered.

Additionally, our modifications to `alive-tv` are completely backwards compatible, ensuring no disruption for tools currently utilizing the Alive2 framework. When a user supplies a single unroll bound our modified `alive-tv` interprets this as a global bound and simply applies this bound as a local unroll factor for all loops in the program. If a user provides more than a single unroll factor and this does not match the number of loops in the program, `alive-tv` will exit and complain.

To ensure the aforementioned points we have also tested our modified `alive-tv` with the Alive2 test pipeline and ensure that all tests pass.

At the time of this writing, our local unroll bounds feature is in the process of becoming upstreamed into Alive2: https://github.com/AliveToolkit/alive2/pull/1300.

4 Experimental Evaluation

We implement the search procedure presented in Figure 1 with Python. The SOLVER takes in a set of constraints on the local unroll bounds which are dispatched to the Z3 SMT-solver for a satisfying assignment.

The elements of $P^{\sqsubseteq}_{k^n}$, which are the nodes used in Algorithm 1, are implemented as Python integer tuples. For each candidate node returned by the solver we invoke our modified version of `alive-tv` with the corresponding local unroll factors. For example, to evaluate the candidate node

$p'_{(1,3)} \sqsubseteq p_{(1,3)}$ in Figure 1 we would invoke `alive-tv` like `alive-tv -src-unroll=1,3 -tgt-unroll=1,3 p p'`.

We evaluate our modified version of `alive-tv` and our search procedure on a modified version of the Test Suite for Vectorizing Compilers (TSVC) [17]. TSVC was originally designed to test the performance of vectorizing compilers. The tests each have timing function calls and helper functions to generate input data and check the output produced by the program. Since we are only performing a static analysis for our evaluation, these helper functions added unnecessary complexity to the verification queries, so we modify the benchmarks to remove these helper function calls that are used at runtime, preserving only the bulk computation of the vectorized kernels.

During testing, we encountered some functions from TSVC where the number of loops did not match after running an LLVM `opt -O3` optimization pipeline. We remove these tests from our evaluation. Additionally, a few tests involve unsupported instructions for `alive-tv` and we also remove these. We are able to evaluate 119 functions from TSVC out of the total 138 functions. The results are presented in Figure 2.

To augment the TSVC benchmark, we also evaluated our strategy on functions from the single-file benchmarks bzip2, gzip, oggenc [18], and SQLite 3.53.0 [26]. No verification coverage improvements were observed for bzip2 and gzip mainly due to timeouts occurring irregardless of the loop unroll bounds used; so these results are not presented. In oggenc and sqlite, there were 86 and 56 functions, respectively, that contained between two and five loops and had matching loop counts across program pairs after optimization. We observed limited cases of a verification coverage improvement for these benchmarks and these results are also presented in Figure 2.

We perform all experiments on a 128 core AMD EPYC 7702P server with 128GB of memory running Ubuntu 24.04. For all compilation tasks we use LLVM version 21.1.8. For `alive-tv` we set an individual SMT query timeout parameter of 10 seconds and allow the solver to use 4GB of memory. We set the `alive-tv` passes parameter to `-O0` and only perform translation validation between pairs of programs run through LLVM `opt` with the `O0` and `O3` pass pipelines. We multi-thread our experiments so the lattice search procedure can be performed for multiple functions in parallel, while the search procedure for each lattice is still single-threaded.

4.1 Effectiveness of Local Unroll Bounds

We present the two main metrics used in Figure 2 to evaluate the effectiveness of our local unroll bounds over the existing global unroll bounds available with `alive-tv`.

Query Ratio. As mentioned in Section 2.2, there can be a wide range of total verification queries required to cover all elements in the lattice. In the best case, we may only need one query to mark all elements. These correspond to verifying $\top$ or encountering a timeout for $\bot$. While in the worst case we may need k^n queries. To gauge the total number of verification queries performed with our random sampling we define the *query ratio* of an explored lattice as the total number of verifier queries to the total number of nodes in the lattice (which is k^n). A smaller query ratio is favorable, as it means only a small portion of the total local unrolling assignment had to be dispatched to the verifier to saturate the entire lattice. The query ratio we measure is directly impacted by the random sampling procedure of lattice nodes.

Coverage Increase. To evaluate the verification coverage improvements local unroll bounds provide over global unroll bounds we define a metric measuring the number of unique local unrolling assignments verifiable in our lattice that are not in the lower closure of any global unrolling assignment inside the lattice. Formally, given an explored lattice $(P^{\sqsubseteq}_{k^n}, \preceq)$ there exist exactly k elements of the lattice which can be verified only with global unroll bounds. These are elements where each local unroll bound is exactly the same. Refer to these elements as $P^{\sqsubseteq}_{(1,..,k)^n}$. Let $p'_{i^n} \sqsubseteq p_{i^n}$ be the top most verified element of $P^{\sqsubseteq}_{(1,..,k)^n}$. Notice that $p'_{i^n} \sqsubseteq p_{i^n}$ is the most precise refinement we can verify with global unroll bounds. The refinement nodes strictly verified with local unroll bounds are given by counting the number of verified nodes in $(P^{\sqsubseteq}_{k^n}, \preceq) \setminus \downarrow (p'_{i^n} \sqsubseteq p_{i^n})$.

We define the *coverage increase* provided by local unroll bounds as the number of verified nodes in $(P^{\sqsubseteq}_{k^n}, \preceq) \setminus \downarrow (p'_{i^n} \sqsubseteq p_{i^n})$ over $|\downarrow (p'_{i^n} \sqsubseteq p_{i^n})|$. A higher coverage increase means more nodes were verified over the best possible verification coverage achievable with global unroll bounds, highlighting the benefits local unroll bounds can provide over global unroll bounds.

Results. In Figure 2 we present the functions from TSVC, oggenc, and sqlite where we observed a coverage increase. For these tests we used a local unroll upper bound of $k = 15$ and only evaluated functions containing two to five loops ($2 \leq n \leq 5$). In general, the random node selection policy used to explore the lattice ensured that the query ratio remained mostly within 10%. The savings provided by randomly selecting nodes in the lattice becomes more apparent as the number of loops increase because the total number of nodes grow exponentially. We observed that the number of verified and timeout queries do not necessarily grow exponentially like the total number of nodes, indicating that that the random node selection policy enables useful pruning of the lattice.

The coverage increase presented in Figure 2 also indicates that we can achieve anywhere from a few percent to over 100% more verification coverage from adopting local unroll bounds. One example presented is for the code in Figure 3.

During our lattice exploration the top most verified element of $P^{\sqsubseteq}_{(1,..,15)^3}$ was $p'_{(10,10,10)} \sqsubseteq p_{(10,10,10)}$, meaning with a global unroll bound we can verify `s114` up to an unroll

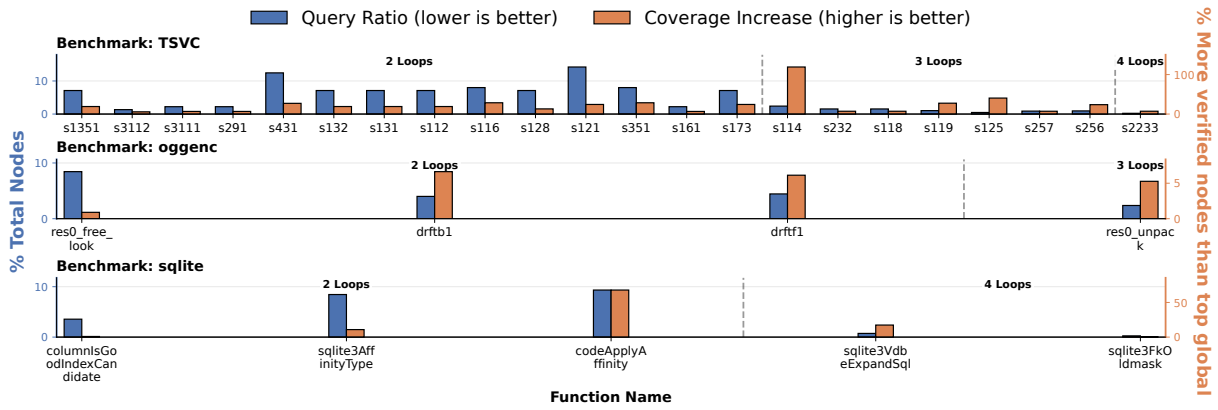

Figure 2. Verification coverage results for functions from TSVC, oggenc, and sqlite. An unroll upper bound of $k = 15$ is used and only functions containing two to five loops are evaluated.

```
for (int nl = 0; nl < 200*(iterations/(LEN_2D)
); nl++) {
    for (int i = 0; i < LEN_2D; i++) {
        for (int j = 0; j < i; j++) {
            aa[i][j] = aa[j][i] + bb[i][j];
}    }    }
```

Figure 3. Loops from the TSVC function `s114`.

bound of ten as anything greater caused a timeout. However, with local unroll bounds we are able to verify elements of the lattice such as $p'_{(15,15,6)} \sqsubseteq p_{(15,15,6)}$ and $p'_{(15,2,15)} \sqsubseteq p_{(15,2,15)}$. The only way we could verify these elements using global unroll bounds would be to increase the TV time out in order to verify $p'_{(15,15,15)} \sqsubseteq p_{(15,15,15)}$ which may be intractable given such an unrolling would copy the loop body a total of 3375 times.

5 Related Work

Bounded model checkers such as the C Bounded Model Checker (CBMC) [10] and ESBMC [6] support manually provided local loop unwinding bounds. The use of *k-induction* has been explored for proving the properties of unbounded loops [5, 7, 20]. However, these approaches have primarily been applied to general property-based verification rather than the specific problem of equivalence checking, which can be modeled in a bounded model checking context using product programs [2, 3]. Exploring the application of these approaches to bounded translation validation remains as future work, although it has been explored for the limited cases of unbounded translation validation [28].

Another approach for scaling equivalence checking is using abstraction refinement. Badihi et al. [1] presented ARDiff, which abstracts syntactically equivalent pieces of code for program pairs as uninterpreted functions (UFs) which can reduce the complexity of encoding the program and checking for refinement. This approach may be useful in the context of bounded translation validation as syntactically equivalent loops can be abstracted in this manner to enable simpler verification conditions.

Lopes et al. [16] also explored the use of UFs to replace loops in programs for equivalence checking. Their approach precisely summarizes loops by computing the closed-form solutions of loops modeled as recurrences. As a consequence, their approach does not support loops with branches and cannot support nested loops where the number of iterations in the inner nest depends on the outer loop.

Purandare et al. [23] used the same idea of searching through a lattice in the context of model checking but their lattice elements are candidate formulae that are checked to hold for a fixed model, with the goal of computing the strongest such formula. They prune candidate elements in their lattice according to the satisfiability and closure of some element. This is very similar to how we prune upper closures when there is a timeout and prune lower closures when some element is verified.

6 Conclusion

This paper presents a formalization of locally bounded program pairs in the context of translation validation along with a formalization of the lattice structure formed by these locally bounded programs. We demonstrate a strategy to search and prune elements in this exponentially sized lattice and show that this approach improves verification coverage for various functions in the TSVC, oggenc, and sqlite benchmarks. We implement our approach by extending Alive2 to support local unroll bounds and our our results indicate that local unroll bounds can be a useful tool for improving bounded verification confidence when the verifier would otherwise timeout.

References

[1] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 13–24. doi:10.1145/3368089.3409757

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *International Symposium on Formal Methods*. Springer, 200–214. doi:10.1007/978-3-642-21437-0_17

[3] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, 100–114. doi:10.1109/CSFW.2004.1310735

[4] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1027–1040. doi:10.5281/zenodo.2646720

[5] Alastair F Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 2011. Software verification using k-induction. In *International Static Analysis Symposium*. Springer, 351–368. doi:10.1007/978-3-642-23702-7_26

[6] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. 2018. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 888–891. doi:10.1145/3238147.3240481

[7] Mikhail YR Gadelha, Hussama I Ismail, and Lucas C Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *International journal on software tools for technology transfer* 19, 1 (2017), 97–114. doi:10.1007/s10009-015-0407-9

[8] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. 2020. Counterexample-guided correlation algorithm for translation validation. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29. doi:10.1145/3428289

[9] Alborz Jelvani. 2026. Alive2-custom. https://github.com/jelvani/Alive2-custom. GitHub repository.

[10] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C Bounded Model Checker: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391. doi:10.1007/978-3-642-54862-8_26

[11] Jay P. Lim, Vinod Ganapathy, and Santosh Nagarakatte. 2017. Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch?. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security* (Dallas, Texas, USA) *(PLAS '17)*. Association for Computing Machinery, New York, NY, USA, 37–42. doi:10.1145/3139337.3139343

[12] Jay P Lim and Santosh Nagarakatte. 2019. Automatic Equivalence Checking for Assembly Implementations of Cryptography Libraries. In *International Symposium on Code Generation and Optimization*. doi:10.5281/zenodo.2229779

[13] Nuno Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 22–32. doi:10.1145/2813885.2737965

[14] Nuno Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2018. Practical Verification of Peephole Optimizations with Alive. In *Research Highlights sections of the Communications of the ACM*. doi:10.1145/3166064

[15] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79. doi:10.1145/3453483.3454030

[16] Nuno P Lopes and José Monteiro. 2016. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer* 18, 4 (2016), 359–374. doi:10.1007/s10009-015-0366-1

[17] Saeed Maleki, Yaoqing Gao, Maria J Garzarán, Tommy Wong, and David A Padua. 2011. An evaluation of vectorizing compilers. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 372–382. doi:10.1109/PACT.2011.68

[18] Stephen McCamant. n.d.. Large single compilation-unit C programs. https://people.csail.mit.edu/smcc/projects/single-file-programs/. MIT Computer Science and Artificial Intelligence Laboratory (CSAIL).

[19] David Menendez and Santosh Nagarakatte. 2017. ALIVE-INFER: Data Driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 49–63. doi:10.1145/3062341.3062372

[20] Jeremy Morse, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. 2013. Handling Unbounded Loops with ESBMC 1.20: (Competition Contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 619–622. doi:10.1007/978-3-642-36742-7_47

[21] George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94. doi:10.1145/358438.349314

[22] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg Berlin, Heidelberg, 151–166. doi:10.1007/BFb0054170

[23] Mitra Purandare, Thomas Wahl, and Daniel Kroening. 2009. Strengthening properties using abstraction refinement. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1692–1697. doi:10.1109/DATE.2009.5090935

[24] Hanan Samet. 1975. *Automatically proving the correctness of translations involving optimized code*. Stanford University.

[25] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 391–406. doi:10.1145/2509136.2509509

[26] SQLite Development Team. 2025. The SQLite Amalgamation. https://sqlite.org/amalgamation.html.

[27] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K Lahiri. 2025. Llm-vectorizer: Llm-based verified loop vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. 137–149. doi:10.1145/3696443.3708929

[28] Anna Zaks and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods*. Springer, 35–51. doi:10.1007/978-3-540-68237-0_5

Received 2026-03-10; accepted 2026-04-15