Towards General Loop Invariant Generation: A Benchmark of Programs with Memory Manipulation

Anonymous Author(s)
Affiliation
Address
email

1 Overview of Supplementary Material

Due to the page limit in the submitted paper, we shall provide more detailed information on our proposed benchmark dataset LIG-MM and the our proposed framework LLM-SE. The supplementary material is organized as follows:

- Sec. 2: Dataset Accessibility and Documentation.
- Sec. 3: Program Example in LIG-MM.
- Sec. 4: Licenses.
- Sec. 5: Details of LLM-SE Framework.
- Sec. 6: Prompt Design for GPT
- Sec. 7: Related Work.
- Sec. 8: Discussion of Limitation, Social Impact, and Outlook.

2 Dataset Accessibility and Documentation

Dataset Documentation: We have documented our dataset for intended researchers as required. The website of our benchmark dataset is available at the following link: `https://anonymous.4open.science/r/LIG-MM-NeurIPS24/`, which includes the programs collected from various sources, the format detail of examples and the code to reproduce the results in our experiments. The link to download the models after fine-tuning is `https://mega.nz/file/M9FEWCjD#QkAQLu7UERPk4Xgb-Rer4U7lfKy7P3rdQeY_p-b8nhM`.

Dataset Statistics: As we mentioned in our paper, the benchmark programs in existing papers mostly contain numerical programs. To fill the lack of benchmarks for general loop invariant generation, we propose LIG-MM, a loop invariant generation benchmark of memory manipulation programs. Table 1 below shows the basics of the code in LIG-MM. Our programs come from four main sources: course codes, competition codes, previous relevant work, and the actual system codes. The programs are modified into a unified format for better usage. Multiple examples are shown in Sec. 3, and the licenses of benchmarks can also be found in Sec. 4.

- *Course codes.* The course code is mainly derived from homework programs on the data structure course and programming language course. The detailed course number and college name are covered due to the anonymity of this paper. These programs contain the most diverse data structures and multi-level pointer operations among the sources of our benchmark.

Submitted to the 38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks. Do not distribute.

Table 1: Statistics of our proposed LIG-MM benchmark.

	Concrete Resources	Number of Programs	Data Structure Types
Course codes	Course homework programs	187	sll, dll, tree, hash-table
Competition codes	SV-COMP [1, 2]	27	sll, dll, tree, hash-table
Previous benchmark	SLING [3, 4]	15	sll, dll, tree
Real-world programs	Linux Kernel [6]	23	sll, dll, hash-table
Real-world programs	GlibC [5]	13	dll, hash-table
Real-world programs	LiteOS [7]	12	dll
Real-world programs	Zephyr [8]	35	sll, dll, hash-table
Overall	-	312	sll, dll, tree, hash-table

- *Competition codes.* SV-COMP[1, 2] is a competition on software verification, which provides a benchmark for verification tasks for comparing verification tools. Originating from competition, this dataset encompasses various verification tasks, providing a comprehensive set of real-world and synthetic examples for testing the effectiveness and efficiency of verification techniques. In our LIG-MM, we select programs from the 2022 competition benchmark.
- *Previous relevant work.* SLING [3, 4] uses traditional dynamic analysis techniques to generate invariants. Other than loop invariants, SLING can also generate preconditions and post-conditions for function. Therefore, not all their benchmarks include the inference of loop invariant or even contain a loop (they use function calls to replace loops). After selection, we choose some of the programs in its benchmark and turning them into a uniform code style.
- *Real-world system codes.* To make the data in LIG-MM closer to real-world software environments, we decide to select more programs from several well-known software and systems. Among them, GlibC [5] is the GNU implementation of the C standard library, providing essential functionalities for numerous applications. Additionally, we have incorporated programs from the Linux Kernel [6], a widely used and highly-regarded operating system kernel that serves as the foundation for countless devices and systems worldwide. To further enhance the diversity of our dataset, we have included LiteOS [7], a lightweight operating system designed for IoT devices, and Zephyr [8], another versatile operating system known for its applicability in resource-constrained environments.

By integrating these varied sources, LIG-MM captures a broad spectrum of programming practices and challenges and ensures that our benchmark is robust and representative of the complexities encountered in multiple scenarios, such as real-world software development and verification. Unlike the numerical program benchmark in previous works [9, 10, 11, 12, 13, 14], our benchmark does not contain pure numerical procedures, all of our programs are related to at least one certain data structure. The data structures we have selected include singly linked lists(sll), doubly linked lists(dll), trees, and hash-tables. In addition, our benchmark includes the usage of multi-level pointers and various pointer arithmetic.

Long-term Preservation and Maintenance: To ensure the longevity and relevance of our proposed LIG-MM benchmark, we will maintain a dedicated public repository on GitHub, facilitating easy access and version control. Regular updates will be made to incorporate new programs, improvements, and community contributions. We encourage open-source dataset, benchmark and codes, inviting researchers to contribute and review submissions to ensure quality. Comprehensive documentation will guide users on structure, usage, and contributions. Engaging with the research community through workshops and conferences will help gather feedback for continuous enhancement. Through these measures, we aim to provide a sustainable resource for ongoing advancements in program verification, loop invariant generation, and memory manipulation analysis.

Terms of Use and License: We have chosen the GPL-2.0 license for our benchmark dataset, and the detailed license is clearly stated on our dataset website.

Discussion of Personally Identifiable Information. As we mentioned before, the course code of our benchmark is mainly derived from homework programs on the data structure course and programming language course. The detailed course number and college name are covered to avoid the link of privacy. Thus, we can confirm that our LIG-MM benchmark does not contain personally identifiable information or offensive content.

3 Program Example in LIG-MM

3.1 Doubly Linked List Example

```
struct list_t {
    struct list_t *prev;
    struct list_t *next;
};

/*@ Let dlistrep(l, p) = l == 0 && emp ||
      ∃ t, data_at(field_addr(l, next), t) *
                data_at(field_addr(l, prev), p) *
                dlistrep(t, l)
 */

/*@ Let dlseg(x, xp, yp, y) = x == y && xp == yp && emp ||
      ∃ z, data_at(field_addr(x, next), z) *
                data_at(field_addr(x, prev), xp) *
                dlseg(z, x, yp, y)
 */

struct list_t *iter_back(struct list_t *l, struct list_t *head)
/*@ With l_prev
    Require dlseg(head, 0, l_prev, l) * dlistrep(l, l_prev)
    Ensure  dlistrep(__return, 0)
 */
{
    struct list_t *p;
    if (l == 0) {
      return head; //@ dlseg(head,0,l_prev,0) * dlistrep(0,l_prev)
    }
    else {
      p = l; //@ l == p && dlseg(head, 0, l_prev, l) * dlistrep(l, l_prev)
      while (p != head) {
        p = p → prev;
      }
      return p;
    }
}
```

This code is changed from function list_for_each_prev of GlibC. This code traverses the entire doubly linked list along the prev pointer. Similar with singly linked list, we define `dlistrep(x,y)` to represent a doubly linked list that starts with x, where the prev of x is y (if x is not 0), `dlseg(x,xp,yp,y)` to represent a segment of doubly linked lists that starts with x and end with yp, where the prev of x is xp and the next of yp is y.

```
S0 : l == p && dlseg(head, 0, l_prev, l) * dlistrep(l, l_prev)
S1 : l != head &&
     p == l_prev && p→next == l &&
     dlseg(head,0,p→prev,p) * dlistrep(l,l_prev)
S2 : l != head && l_prev != head &&
     p→next == l_prev && l_prev → next == l && l_prev → prev == p &&
     dlseg(head,0,p→prev,p) * dlistrep(l,l_prev)
S3 : ∃ p0, l != head && l_prev != head && p0 != head &&
     p0→next == l_prev && l_prev → next == l &&
     l_prev → prev == p0 && p0 → prev == p && p → next == p0 &&
     dlseg(head,0,p→prev,p) * dlistrep(l,l_prev)

One valid loop invariant:
    l == p && dlseg(head, 0, l_prev, l) * dlistrep(l, l_prev) ||
    dlseg(head,0,p→prev,p) * dlseg(p→next,p,l_prev,l) * dlistrep(l,l_prev)
```

With symbolic execution, we can observe that p divides `dlseg(head,0,l_prev,l)` into two segments, so we can guess `dlseg(head,0,p→prev,p) * dlseg(p→next,p,l_prev,l)` and get one valid loop invariant.

3.2 Singly Linked List with Multi-level Pointers Example

```
struct list {
    struct list *tail;
};

/*@ Let listrep(l) = l == 0 && emp ||
      ∃ t, data_at(field_addr(l, tail), t) * listrep(t)
 */

/*@ Let lseg(x, y) = x == y && emp ||
      ∃ z, data_at(field_addr(x, tail), z) * lseg(z, y)
 */

/*@ Let listbox_rep(x) = ∃ p, *x == p && listrep(p) */


/*@ Let listbox_seg(x,y) = x == y && emp ||
       ∃ p, *x == p && listbox_seg(&(p→tail),y)
*/

struct list ** malloc_list(void)
  /*@ Require emp
      Ensure ∃ p, *__return == p && emp
   */
  ;

void free_list(struct list * * p2)
  /*@ With p
      Require *p2 == p && emp
      Ensure emp
   */
  ;

struct list *iter(struct list *x)
  /*@ Require listrep(x)
      Ensure listrep(__return)
  */
{
  struct list * * t, * * px;
  px = malloc_list();
  t = px;
  * t = x;
  /*@ t == px && *t == x && listrep(x) */
  while ( * t != (void *) 0) {
    t = &(( *t) → tail);
  }
  x = * px;
  free_list(px);
  return x;
}
```

This code uses second-order pointers to traverse a singly linked list, and in addition to `listrep` and `lseg`, we also introduce `listbox_rep` and `listbox_seg` to represent the corresponding second-order pointer structure.

```
S0 : t == px && *t == x && listrep(x)
S1 : x != 0 &&
     *px == x && *t == x → tail && listrep( *t )
S2 : ∃ p0, x != 0 && p0 != 0 &&
     *px == x && x → tail == p0 && *t == p0 → tail && listrep( *t)
S3 : ∃ p0 p1, x != 0 && p0 != 0 && p1 != 0 &&
     *px == x && x → tail == p0 && p0 → tail == p1 &&
     *t == p1 → tail && listrep( *t)

One valid loop invariant:
     listbox_seg(px, t) * listrep( *t)
```

With symbolic execution, we can observe that the two secondary pointers, px and t, point to a segment of a singly linked list. so we can guess `listbox_seg(px,t)` and get one valid loop invariant.

3.3 Tree Example

```
struct tree {
    int data;
    struct tree * left;
    struct tree * right;
    struct tree * parent;
};

/*@
  Let tree_rep(p, p_par) = p == 0 && emp ||
        ∃ p_lch p_rch,
                        data_at(field_addr(p, left), p_lch) *
                        data_at(field_addr(p, right), p_rch) *
                        data_at(field_addr(p, parent), p_par) *
                        tree_rep(p_lch, p) *
                        tree_rep(p_rch, p)
*/

/*@
  Let ptree_rep(p, p_par, p_root, p_top) = p == p_root && p_par == p_top && emp ||
        ∃ ppar_rch ppar_par ,
                        data_at(field_addr(p_par, left), p) *
                        data_at(field_addr(p_par, right), ppar_rch) *
                        data_at(field_addr(p_par, parent), ppar_par) *
                        tree_rep(ppar_rch, p_par) *
                        ptree_rep(p_par, ppar_par, p_root, p_top) ||
        ∃ ppar_lch ppar_par ,
                        data_at(field_addr(p_par, left), ppar_lch) *
                        data_at(field_addr(p_par, right), p) *
                        data_at(field_addr(p_par, parent), ppar_par) *
                        tree_rep(ppar_lch, p_par) *
```

```
                              ptree_rep(p_par, ppar_par, p_root, p_top)
*/


struct tree *Find_root(struct tree * x)
/*@ With fa root
    Require x != 0 && tree_rep(x, fa) * ptree_rep(x, fa, root, 0)
    Ensure tree_rep(__return , 0)
 */
{
    while (x → parent)
      x = x → parent;
    return x;
}
```

This code traverses the tree from x to root along the parent pointer. We define `tree_rep(x,fa)` to represent a tree with root x, where the parent of x is fa (if x is not 0). And we use `ptree(p, p_par, p_root, p_top)` to represent a part of tree with a hole at p, which means that `tree_rep(p_root, p_top) = tree_rep(p, p_par) * ptree(p, p_par, p_root, p_top)`.

```
S0 : x != 0 && tree_rep(x, fa) * ptree_rep(x, fa, root, 0)
S1 : ∃ x0, x0 != 0 && fa != 0 &&
     x == fa && tree_rep(x0,fa) * ptree_rep(x0,fa,root,0)
S2 : ∃ x0 x1, x0 != 0 && fa != 0 && x != 0 &&
        fa → left == x0 && fa → right == x1 && fa → parent == x &&
        tree_rep(x1,fa) * ptree_rep(fa,x,root,0) * tree_rep(x0,fa) ||
     ∃ x0 x1, x0 != 0 && fa != 0 && x != 0 &&
        fa → right == x0 && fa → left == x1 && fa → parent == x &&
        tree_rep(x1,fa) * ptree_rep(fa,x,root,0) * tree_rep(x0,fa)

One valid loop invariant:
    x != 0 && tree_rep(x, fa) * ptree_rep(x, fa, root, 0) ||
    ∃ x0, x != 0 && tree_rep(x0, x) * ptree_rep(x0, x, root, 0)
```

Since S3 has 4 branches, we've omitted it here. But from S0 S1 S2 we can already find some patterns, we can guess `∃ x0, x!= 0 && tree_rep(x0, x)` and get one valid loop invariant.

4 Licenses

In the components of our LIG-MM, the programs derived from course homework are collected on our own, and the other programs are selected from existing benchmarks. Here, we will list the sources and licenses of the existing benchmarks used in our work.

- SLING [3, 4]. The official website of SLING is `https://github.com/guolong-zheng/sling`, and we used the PLDI version of their benchmark listed in the repository. Currently, there is no license on their website. In our work, we strictly follow their instructions, and we believe there is no risk of infringement.
- SV-COMP [2]. The official website of SV-COMP is `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c`, and it follows the Apache-2.0 license. The SPDX-FileCopyrightText is 2011-2013 Alexander von Rhein, University of Passau and 2011-2021 The SV-Benchmarks Community.
- Linux Kernel [6]. We use the programs from `https://github.com/torvalds/linux/`, which collects the code of the Linux kernel. There are multiple licenses in this repository and the programs we select all follow the GPL-2.0 license (`https://github.com/torvalds/linux/blob/master/LICENSES/preferred/GPL-2.0`).

- GlibC [5]. The official GitHub repository of GlibC is `https://github.com/kraj/glibc/blob/master/`. The license of their code is GNU LESSER GENERAL PUBLIC LICENSE (LGPL) v2.1 `https://github.com/kraj/glibc/tree/master?tab=LGPL-2.1-2-ov-file`.
- LiteOS [7]. The official website of LiteOS is `https://gitee.com/openharmony/kernel_liteos_m` and their license is BSD 3-Clause License `https://gitee.com/openharmony/kernel_liteos_m/blob/master/LICENSE`.
- Zephyr [8]. The official GitHub repository of Zephyr is `https://github.com/zephyrproject-rtos/zephyr`, and it follows the Apache-2.0 license (`https://github.com/zephyrproject-rtos/zephyr/blob/main/LICENSE`).

For our proposed LIG-MM benchmark dataset, we have chosen the GPL-2.0 license, and the detailed license is clearly stated on our dataset website.

5 Details of LLM-SE framework

As mentioned in the submitted paper, our proposed LLM-SE combines large language models and symbolic execution. Recall the overview in Sec.4 of the submitted paper, LLM-SE includes two processes, the offline self-supervised training process and the online querying process. In the following section, we shall introduce these two processes. We suggest the reader understand Sec.4 of the submitted paper first before reading the following documents.

5.1 Offline Training with Self-supervised Learning

Our methodology relies on the usage of LLM, which is essential for solving the complex task of loop invariant generation. In practice, LLM after pretraining on large corpora of text can be fine-tuned on specific tasks and domains. We can leverage its powerful language understanding and generation abilities in loop invariant generation. After extensive investigation and experiments, we choose one pretrained LLM named CodeGen [15, 16] to propel our approach. It is tailored explicitly for code-related tasks with various applications [17, 18, 19], endowed with a comprehensive understanding of code structures, program semantics, and syntax. The unique prowess of CodeGen lies in its ability to decode and generate code snippets by comprehending the intricacies of different programming languages. Its robust capabilities allow us to delve into the task of loop invariant generation, presenting a promising avenue for addressing the complexities inherent in this task.

After selecting the pretrained LLM, we need to fine-tune it on our task. However, a major challenge is that we lack enough labeled data, i.e., programs with valid loop invariants. Without sufficient data for fine-tuning, the LLM cannot fully demonstrate its potential. To overcome this challenge, we adopt a self-supervised learning approach to generate abundant labeled data for fine-tuning our LLM. By formulating auxiliary tasks within the self-supervised framework, we generate rich synthetic data by employing a split-and-reassembly technique based on data structure definitions. Such a strategy allows us to produce ample training data, which is essential for fine-tuning LLM in the absence of labeled data, and thus addresses the challenge of data scarcity.

The detailed process of our self-supervised learning paradigm is shown in Figure 1. In this figure, we examine the "splitter" module of Figure 3 of the submitted paper , as the part enclosed by the green dash line. Given the data structures defined by the users, every time we sample one predicate from the data structures, e.g. `lseg`. Then, we check its definitions given by the users shown in the purple box:

```
lseg(x,y) = x == y && emp || ∃ z, x→tail == z * lseg(z,y)
```

We can see that there are two branches in its definition, one for the empty case and the other one for the non-empty case. We find that multi-branch definition is very common in data structures. Therefore, we decide to split the predicate and let the LLM try to reassemble it. This *predicate recovery* task is the auxiliary task designed for fine-tuning the LLM. Back to the purple box in the

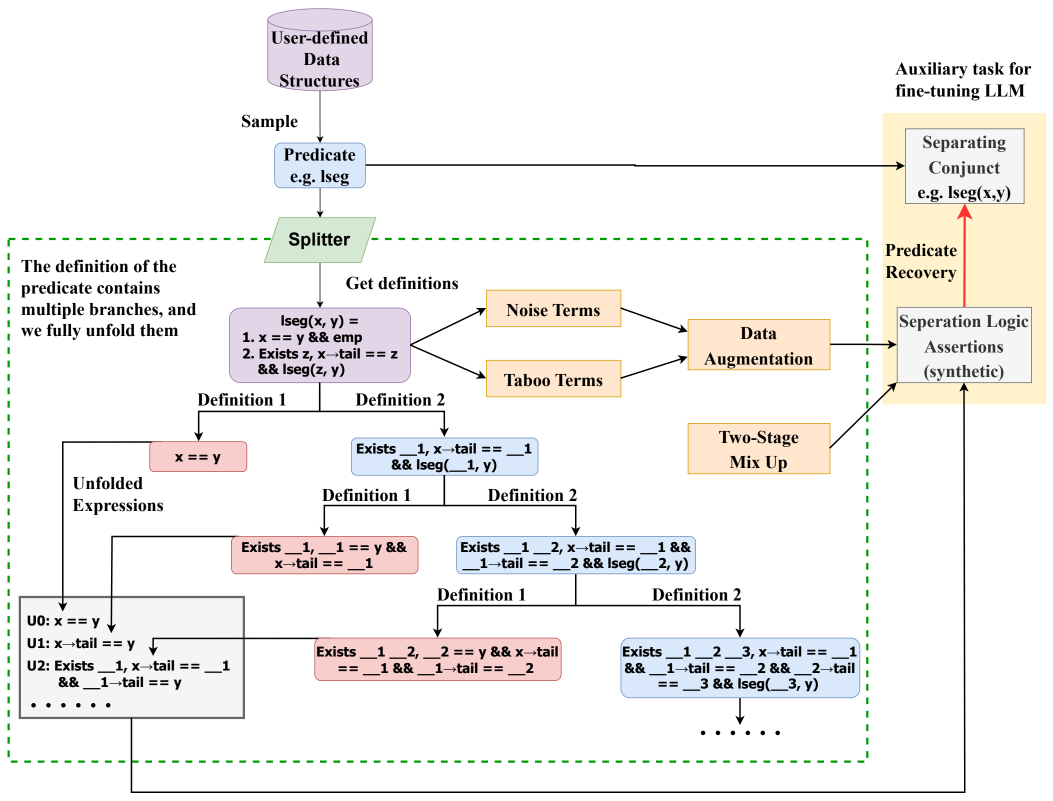
User-defined Data Structures
Sample
Predicate e.g. lseg
Auxiliary task for fine-tuning LLM
Separating Conjunct e.g. lseg(x,y)
Predicate Recovery
Seperation Logic Assertions (synthetic)
Splitter
The definition of the predicate contains multiple branches, and we fully unfold them
Get definitions
lseg(x, y) =
1. x == y && emp
2. Exists z, x→tail == z && lseg(z, y)
Noise Terms
Taboo Terms
Data Augmentation
Two-Stage Mix Up
Definition 1
Definition 2
x == y
Exists __1, x→tail == __1 && lseg(__1, y)
Unfolded Expressions
Exists __1, __1 == y && x→tail == __1
Exists __1 __2, x→tail == __1 && __1→tail == __2 && lseg(__2, y)
U0: x == y
U1: x→tail == y
U2: Exists __1, x→tail == __1 && __1→tail == y
• • • • • •
Exists __1 __2, __2 == y && x→tail == __1 && __1→tail == __2
Exists __1 __2 __3, x→tail == __1 && __1→tail == __2 && __2→tail == __3 && lseg(__3, y)
• • • • • •

Figure 1: Our self-supervised learning paradigm for fine-tuning LLM. To solve the data scarcity issue, we design an auxiliary task called *predicate recovery*. We split the data structure completely based on its definitions, further process them from unfolded expressions to synthetic separation logic assertions, and let the LLM try to recover the original separating conjunct. Moreover, we apply multiple data augmentation and mix up strategies to refine the synthetic assertions to mimic the real ones, making the auxiliary task more challenging.

figure, we begin to recursively split `lseg(x,y)` based on these two branches, shown as the expanding arrows with "Definition 1" and "Definition 2". We noticed that the expressions after "Definition 1" expansion lead to an end of further expansions, as the predicate `lseg` itself has been eliminated. We mark these expressions with red boxes and the others with blue boxes. For the expressions in blue boxes, we can further expand them based on these two branches recursively. Repeatedly, we can acquire an expansion tree as shown in the figure.

After splitting the predicate, we select the expressions in the red boxes, the terminating nodes of the expansion tree for further usage. Then, we further process them to remove the redundant temporal variables. For example, the temporal variable `__2` in the bottom red plays no significant role and can be directly replaced by program variable `y`. After the processing, we can obtain the synthetic unfolded expressions, as shown in the grey box at the left bottom corner of the figure. In this way, we successfully split the predicate into a set of unfolded expressions. We define this process as `Gen0(C)`, such as the example in the figure:

```
Gen0(lseg(x, y)) = [U0, U1, U2, ...]
```

Admittedly, there is still a gap between the generated unfolded expressions `[U0,U1,U2,...]` and the real separation logic assertions, where the real assertions may contain other separating conjuncts or pure proposition. To address this issue, we further employ two techniques in self-supervised learning: *data augmentation* and *mix up*. Data augmentation [20, 21] involves modifying the data by applying various transformations, such as rotation, scaling, or flipping, creating augmented versions of the original data. These altered examples offer a more diverse range of inputs for the model, helping it to generalize to unseen data. Mix up [22, 23], on the other hand, is a regularization technique that blends samples in the dataset, combining two examples by interpolating their features and labels. By averaging two data samples, mix up encourages smoother decision boundaries, mitigating overfitting and enhancing the ability to learn from diverse data. Both data augmentation and mix up are instrumental in increasing the diversity and variability within the data, thereby boost the capacity to handle real-world scenarios. Next, we shall introduce how we adapt data augmentation and mix up in our work.

Data augmentation. Based on the generated unfolded expressions `[U0,U1,U2,...]`, we try to augment them by adding more terms. Referring to Figure 1, we can see there are two right arrows from the purple box. They indicate that we derive noise terms and taboo terms based on the definition of the predicate. Taking `lseg(x,y)` for an example: based on its definition, we know it is an manipulation of single list and contains operations to its `tail` field. Therefore, the possible noise terms to add for this separating conjunct are:

```
{}=={}, {}!={}, listrep({}), {}→tail=={}, listrep(y), y→tail=={}
```

where `{}` could be program variables, temporal variables, or even `NULL`. Since the definition of `lseg(x,y)` already includes the memory address operation of `x`, more description to this memory is not allowed. Therefore, `x→tail={}, listrep(x)` are regarded as the taboo terms of `lseg(x,y)`. After specifying the noise terms and taboo terms, we can randomly add more terms to the original separation expressions based on a set of certain probabilities, and we defined this process as `Gen1(C) = augment(Gen0(C))`, for example:

```
Gen1(lseg(x, y)) = augment(Gen0(lseg(x, y)) = augment([U0, U1, U2, ...])
= [U0 && y != 0 && ..., U1 && y→tail==0 && ..., U2 && listrep(y) && ..., ...]
```

Mix up. To further enhance the quality, we design a two-stage mix up strategy. Suppose we have split the predicates `A`, `B`, and obtain their unfolded expressions `[A0,A1,A2,...]`, `[B0,B1,B2,...]`. We have two ways to mix up their predicates and expressions. In the first way, we mix their predicates as `A*B`. The corresponding expressions shall become `[A0*B0,A1*B1,A2*B2,...]`. It can mimic the situation that the loop invariant contains multiple predicates. In the second way, we combine their predicates `A||B`. The combined expressions will be `[A0||B0,A1||B1,A2||B2,...]`. It is very common for loop invariants to contain multiple terms separated by `||`, especially for programs in the real world. Our two-stage mix up is conducted recursively with a certain probability, which results in a diverse and changeable combination of expressions. We define the aforementioned process as `Gen2(C)`, where `C` is composed of one or multiple separating conjuncts.

```
Gen2(C) = Gen1(C) if C is a single conjunct,
          Gen2(A) * Gen2(B) if C = A * B,
          Gen2(A) || Gen2(B) if C = A || B
```

In the end, we can obtain a batch of synthetic separation logic assertions more close to real assertions, which can be used to create the *predicate recovery* task. In this auxiliary task, the synthetic separation logic assertions are regarded as input and the original separating conjunct is regarded as output or the label. As this whole process is fully automated, we can easily generate rich labeled data and fine-tune our LLM with this auxiliary task.

5.2 Online Querying with Interactive System

Following the offline training, we obtain a LLM that is capable of inferring the separating conjuncts given the assertions. Building on this LLM and the conventional techniques of symbolic execution and entailment solver, we devise an interactive framework that can generate loop invariants online by multiple interactions. In this section, we present the loop invariant generation process for the single-layer loop case, and then demonstrate how to generalize it to the multi-layer loop case.

Algorithm 1: Single_loop_inv_gen

```
Data: Pre : precondition
      e : an expression of loop condition
      body : a loop-free program fragment
Result: inv : loop invariant for program c = while (e) {body}
1  I[0] = Pre
2  for i : 0 → MAX_NUM - 1 do
3  |   I[i+1] = Symbolic(true_condition(I[i],e), body)
4  end
5  invs = Infer_invs(I)
6  inv = Pick_invs(I,invs)
7  inv_post = Symbolic(true_condition(inv,e), body)
8  if entailment_solver(inv, Pre) && entailment_solver(inv, inv_post) then
9  |   return inv
10 end
11 else
12 |   exit("LLM Inference Fail")
13 end
```

5.2.1 Basic Interaction of Single Loop

When focusing on single loop programs, we assume that the loop is not preceded or followed by any statements. We take the loop condition `e`, the loop body `body` and the loop precondition `Pre` as input, as shown in Algorithm 1. We perform a number of symbolic executions and obtain the postcondition array `I` for executing the loop body $0 \sim$ **MAX_NUM** times. The function `true_condition` computes the postcondition of the precondition when the loop condition `e` holds, ensuring that the loop body can be entered. We then obtain the loop invariant inferences `invs` from LLM using the function `Infer_invs`. Then, we will obtain a simplified invariant assertion `inv` by the function `Pick_invs` based on the calculated postcondition sequence `I` and the returned invariant sequence `invs`. Finally, we verify the correctness of the loop invariant by `entailment_solver` via our entailment solver. If the loop invariant is correct, we return it.

The key point of the algorithm introduced above is to obtain enough information by multiple symbolic executions and output invariants according to the calculated postconditions. The former has already been mentioned before, while the latter relies on the interaction between our LLM and traditional tools. In each round of interaction, we partition the assertion set `I` into two subsets `succ_I` and `fail_I` based on the separating conjunct `sep` inferred by the well-trained LLM. The subset `succ_I` consists of the assertions that are successfully replaced by `sep`, while the subset `fail_I` comprises the assertions that are not successfully replaced.

```
∀ i ∈ succ_I, i ⊢ sep * True
∀ i ∈ fail_I, i ⊬ sep * True
```

Subsequently, we apply repeated operations on the two subsets `succ_I` and `fail_I` , and eventually obtain the two invariant results `inv_1` and `inv_2`, which are combined as the solution for `I`. If the assertions in the set `I` are deemed to be similar assertions, we directly return the new assertion that is derived from the similar assertions as the solution.

Taking list reverse program as an example, we assume that `I=[S0,S1,S2,S3]`, and suppose that LLM returns `lseg(w,p)` in the first round of interaction. At this point, we have `succ_I = [S1',S2',S3']` and `fail_I = [S0]`.

```
S1' : v == t && p → tail == 0 && lseg(w,p) * listrep(v)
S2' : v == t && p → tail == 0 && lseg(w,p) * listrep(v)
S3' : v == t && p → tail == 0 && lseg(w,p) * listrep(v)

inv_1 :  p → tail == 0 && lseg(w,p) * listrep(v)
```

Then we perform the second round of interaction on `succ_I` and `fail_I` separately. In the recursive process of `succ_I`, we find that `S1'`, `S2'` and `S3'` are similar assertions, so we return `inv_1`. In the

Algorithm 2: Infer_invs

```
Data: I: list of calculated postcondition
Result: inv : invariant inferred from I
if I == [] then
    return ""
end
else
    if Similar(I) then
        return Similar_extract(I)
    end
    else
        sep = use_llm(model, I)
        succ_I, fail_I = use_solver(I, sep)
        inv_1 = Infer_invs(succ_I)
        inv_2 = Infer_invs(fail_I)
        inv = inv_1 || inv_2
        return inv
    end
end
```

recursive process of `fail_I`, we find that there is only `S0` in the set, so we directly return `inv_2 : S0`. Therefore, the final answer is `p` $\to$ `tail == 0 && lseg(w,p) * listrep(v) || S0`.

Algorithm 2 implement the above interaction process. We denote the separating conjunct inference function by `use_llm(model,I)`, which returns the inferred separating conjunct under the given LLM `model` and the separation logic assertion set `I`. The conjunct updating function is `use_solver`, and we classify the assertions according to the results of the entailment solver. Note that we use the `Similar` and `Similar_extract` function in the termination condition judgment. The `succ_I` of the list reverse program above is a typical example. We can directly find that they are the same assertion after performing the string substitution algorithm. If Algorithm 1 fails to find a valid loop invariant, we will mask the current output of LLM and retry it.

For generating the final loop invariant, we can use the disjunction of all assertions as we did in last case, but this way may not be concise. We hope to select the smallest set of assertions that can cover all cases from them (here we use a disjunction of assertion as a set of assertions to select), so we propose a Pick Algorithm to solve this problem. Our Pick Algorithm addresses the fundamental problem of selecting a minimal set of assertions that covers all the assertions. We formulate the problem as a SAT problem for convenience. We associate each assertion with two points, namely the Cover Point that indicates the coverage of the assertion and the Pick Point that indicates the selection of the assertion. We construct relevant clauses based on assertion derivation. We compute the set of assertions S_i that can be entailed by each assertion A_i, that is, A_i implies each element $A_j \in S_i$, where A_i is included in S_i. We then construct a clause: $C_i \to \bigvee P_j$, indicating that any assertion in S_i can cover A_i if selected. Hence, our final solution is the set of assertions with P_i being true under the satisfiability of $\bigwedge C_i$.

5.2.2 Extension to Multi Loop Scenario

In this section, we discuss how to extend our algorithm for single loop programs to the general multi loop case. Before, we assume that the single loop program consists of only one loop statement, and that there are no statements before or after the loop. To handle statements before or after the loop, we simply need to compute the loop body precondition from the program precondition, which is a straightforward step that we omit here. For the case of multiple loops, we focus on nested loops, which are loops that contain other loops inside their body. The case of sequential loops, which are loops that follow each other, can be easily handled by applying the algorithm for single loop programs repeatedly.

Algorithm 3: Multi_loop_inv_gen

```
Data: Pre : precondition
      e : an expression of loop condition
      body : a program fragment
Result: inv : loop invariant for the outer loop
I[0] = Pre
if loop-free body then
    return Single_loop_inv_gen(I[0],e,body)
end
else
    (c_before, inner_e, inner_body,c_after) = Split_program(body)
    for i : 0 → MAX_NUM - 1 do
        inner_pre = Symbolic(true_condition(I[i],inner_e),c_before)
        inner_inv = Multi_loop_inv_gen(inner_pre,inner_e,inner_body)
        I[i+1] = Symbolic(false_condition(inner_inv,inner_e), c_after)
    end
    invs = Infer_invs(I)
    inv = Pick_invs(I,invs)
    inv_post = Symbolic(true_condition(inv,e), body)
    if entailment_solver(inv, Pre) && entailment_solver(inv, inv_post) then
        return inv
    end
    else
        exit("LLM Inference Fail")
    end
end
```

The biggest difficulty in our implementation is to find the strongest postcondition of the outer loop executing a single iteration. Since we do not know the loop invariant of the inner loop, we need to first find the loop invariant `inner_inv` of the inner loop under the current precondition `S0` of the outer loop, and this is a smaller sub-problem. Once we find `inner_inv`, we can perform symbolic execution from `S0` to get `S1`. Similarly we can find `S2`, `S3` and so on, then we can manage to generate the loop invariant of the outer loop. Finally, we only need to infer the loop invariant of the inner loop based on the loop invariant of the outer loop, and then we have completed the generation of all loop invariants of the entire program.

We present the specific algorithm implementation in Algorithm 3. The input parameters are similar to Algorithm 1, the only difference is that the loop body of Algorithm 3 can contain other loops. If `body` does not involve any other loop statements, we invoke Algorithm 1 directly. Otherwise, we will apply the `Split_program` function to decompose it into four components: the inner-pre-loop statement `c_before`, the inner loop condition `inner_e`, the inner loop body `inner_body` and the inner-post-loop statement `c_after`. We compute the pre-loop condition `inner_pre` by symbolically executing `inner_e`. By recursive calling, we can obtain the `inner_inv` for inner loop under the precondition `inner_pre`, which allows us to determine the exit condition of the inner loop. Here we use `false_condition` to computes the postcondition of the precondition when the loop condition `e` does not hold. Based on this, we can calculate the strongest postcondition for continuing to execute `c_after`, which is completing one iteration of body from `I[i]`. The remaining work is similar to Algorithm 1, and we will not go into details here.

6 Prompt Design for GPT

6.1 Prompt Text for GPT-4

When using GPT-4 as the baseline in our experiments, we write a common prompt text, which we add before each program of our benchmark. The prompt text is given as follows, and the examples of predicate definitions and programs can be found in the previous section (Sec. 3).

```
You will receive a program, and please fill out the 'INFILL' parts with
    suitable loop invariants.
Please only output loop invariants and no more text is needed.
You may use the defined predicates below and the data_at operation in your
    invariants.
(Note: data_at operation is an atomic operation, data_at(x, v) denotes that
     the memory location x contains the value v.)
Definitions:
1. xxx
2. xxx
xxx
Program:
xxx
```

7 Related Work

Traditional approaches for program invariant generation rely on program synthesis and analyze. LoopInvGen is a data-driven tool that generates provably sufficient loop invariants for program verification [24]. It transforms the loop invariant inference problem into a sequence of precondition inference problems, and solves them using a precondition inference engine (PIE [25]), which employs a program synthesis technique to learn features in a focused manner. Llinva [26] implemented an algorithm that automatically generates loop invariants using Why3 [27] and GPID [28]. It generates verification conditions using Why3, and strengthens the expressions in the verification conditions using GPID and abduction techniques. Then, it passes them to an SMT solver to check their validity, and repeats this process until it obtains the loop invariants. By analysis the execution process for some test inputs, SLING [3] can automatically generate precondition, post-condition and loop invariants without other messages. But SLING only infers shape properties using inductive predicates and pure equality. These properties have strict patterns and they do not consider general disjunctive invariants or numerical relations. Crucially, it is very dependent on the sample inputs. It need smart test-input generation techniques to keep the correctness and efficiency. [29] adapts a different technique, which generates loop code from the loop invariants to learn specified algebraic relations among their terms, instead of others generating invariants from the loop body.

Thanks to the rapid development of machine learning, several learning-based approaches have been has been increasingly studied. ICE-DT [30] utilizes decision trees over manually designed features, and intuitively uses one learner and one teacher for predicting invariants. CODE2INV [11] uses reinforcement learning (RL) with graph neural networks to train the agent for generating candidate loop invariants following the syntax and semantic. With the help of RL, they do not need labeled data with expert knowledge, while utilizes Z3 [31] to provide reward for training. Though it can outperform LOOPINVGEN and ICE-DT, the main drawback of using RL is that they need to directly train their agents on the evaluation sets with numerous trials and errors, which could lead to inefficiency. Following CODE2INV, several works have been proposed to improve it. [32] combines RL with an heuristic method called nondeterministic strategies, where RL is trained to guide the searching direction of the heuristic method. [33] expands CODE2INV on non-linear loop invariants via well-trained gated continuous logic networks, which improve the performance of CODE2INV significantly. CLN2INV [34] replace the graph neural networks used in CODE2INV to continuous logic networks and introduce a SMT-based tool to make it possible to generate loop invariants for real-world programs. [35] proposes to automatically construct inductive loop invariants for loop structures consisting of multiple paths inside, which generates loop invariants between forward and backward reachability of the loop. However, existing machine learning based methods for loop invariant generation can only handle numerical programs with scalar variables, which limits their usage since programs in the real world often include data structures and memory operations.

Recently, large language models (LLM) have emerged as a transformative force in the field of machine learning, particularly when applied to code-related tasks [15, 16, 36]. The advent of models such as BERT (Bidirectional Encoder Representations from Transformers [37]) and GPT (Generative Pretrained Transformers [38]) has signified a turning point in the way we approach natural language understanding and generation. These models are pretrained on vast corpora of text, such as Wikipedia, books, and news articles. By learning from these diverse sources, LLM acquire a deep understanding of natural language, including its structure, context, and semantics. They use a special architecture called Transformer, which consists of multiple layers of self-attention and feed-forward layers. The Transformer architecture enables LLM to capture long-range dependencies and complex patterns during pre-training. After pretraining on large corpora of text, LLM can be fine-tuned on specific tasks and domains, such as text generation, classification, and summarization. By using LLM, we can leverage their powerful language understanding and generation abilities. This recent surge in LLM development has inspired a fresh wave of research in code-related tasks, ranging from code summarization, code completion, and code-to-code translation.

In our work to explore loop invariant generation, we draw upon the impressive strides made in LLM's ability to process and generate logic assertions, as these models promise to transcend the challenges posed by complex program semantics and syntax. The synergy between LLM and code-related tasks presents an exciting avenue for our proposed approach in this paper. Several recent work [14, 12, 13] also try to utilize LLM to solve the invariant generation task. One possible paradigm is to use traditional solvers such as Daikon [39] to generate valid invariants for the programs in the dataset, and regard the generated invariants as labels for training. In other words, they regard traditional solvers as the oracle and use LLM to approximate it. This approach has a major drawback that the LLM trained by traditional solver can hardly outperform the solver under the same conditions. In contrast, our self-supervised learning approaches LLM-SE do not have this limitation.

To the best of our knowledge, existing machine learning-based methods are restricted to verifying programs' numerical properties, neglecting shape analysis and memory safety verification for real-world programs with diverse data structures and memory manipulation. Therefore, we consider it necessary to propose a new benchmark to cover the programs with memory manipulation, as the LIG-MM benchmark proposed in our work.

8 Discussion of Limitation, Social Impact, and Outlook

While our work represents a significant advancement in the field of loop invariant generation, it is not without its limitations. Firstly, although our proposed LLM-SE framework demonstrates strong performance on various benchmarks, its pass rate is not yet sufficient for seamless integration into all real-world software applications. The reliance on the accuracy and comprehensiveness of separation logic predicates means that any gaps in these definitions can potentially limit the model's effectiveness. Moreover, while our method is designed to generalize across various data structures and multi-loop scenarios, specific edge cases and complex data manipulations remain that may not be fully addressed. The computational overhead associated with symbolic execution and the interactive querying process may also pose challenges for scaling up to very large and complex software systems.

As for the social impacts. On the positive side, enhancing the ability to verify and validate software automatically can lead to more reliable and secure systems. This is particularly crucial in domains such as healthcare, finance, and automotive industries, where software correctness can directly and significantly impact safety and security. By reducing the need for extensive manual intervention in the verification process, our approach can also democratize access to high-assurance software development, enabling smaller teams and organizations to build reliable systems without requiring deep expertise in formal methods. However, there are also potential negative implications to consider.

The automation of verification tasks may lead to job displacement for some roles traditionally involved in software testing and verification. Additionally, as with any AI-driven technology, there is a risk of over-reliance on automated tools, which may result in a false sense of security if the tools are not used properly or if their limitations are not fully understood. Ensuring transparency in how these tools work and maintaining human oversight in critical decision-making processes will be essential to mitigate these risks.

Looking ahead, several promising directions for future research and development exist. One key area is enhancing our LLM-SE framework to further improve its accuracy and applicability. This could involve refining the self-supervised learning paradigm, exploring more sophisticated symbolic execution techniques, and developing more comprehensive sets of separation logic predicates. Additionally, integrating our approach with other program analysis tools and methodologies could provide a more holistic solution for software verification.

Expanding the scope of our benchmarks to include an even wider range of real-world software systems will also be important for validating and improving the robustness of our methods. Collaboration with industry partners could facilitate access to more diverse datasets and real-world use cases, driving further innovation and practical impact.

Finally, fostering a community of researchers and practitioners around loop invariant generation and related areas will be crucial for sustaining progress. By sharing our findings, tools, and datasets openly, we aim to contribute to the collective knowledge and capabilities of the field, encouraging further exploration and refinement of automated software verification techniques.

In summary, while our work marks a significant step forward, it also opens up numerous opportunities for future research and development. By addressing the current limitations, understanding the broader social impacts, and continuing to innovate, we can move closer to realizing the full potential of automated loop invariant generation in creating reliable, secure, and high-assurance software systems.

References

[1] D. Beyer, "Progress on software verification: Sv-comp 2022," in *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II.* Berlin, Heidelberg: Springer-Verlag, 2022, p. 375–402. [Online]. Available: https://doi.org/10.1007/978-3-030-99527-0_20

[2] "Sv-comp benchmark." [Online]. Available: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c

[3] T. C. Le, G. Zheng, and T. Nguyen, "SLING: using dynamic analysis to infer program invariants in separation logic," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 788–801. [Online]. Available: https://doi.org/10.1145/3314221.3314634

[4] "Sling benchmark." [Online]. Available: https://github.com/guolong-zheng/sling

[5] "Source code of glibc." [Online]. Available: https://github.com/kraj/glibc/blob/master/

[6] "Source code of linux kernel." [Online]. Available: https://github.com/torvalds/linux/

[7] "Source code of liteos." [Online]. Available: https://gitee.com/openharmony/kernel_liteos_m

[8] "Source code of zephyr." [Online]. Available: https://github.com/zephyrproject-rtos/zephyr

[9] S. Padhi and T. D. Millstein, "Data-driven loop invariant inference with automatic feature synthesis," *CoRR*, vol. abs/1707.02029, 2017. [Online]. Available: http://arxiv.org/abs/1707.02029

[10] S. Padhi, T. D. Millstein, A. V. Nori, and R. Sharma, "Overfitting in synthesis: Theory and practice (extended version)," *CoRR*, vol. abs/1905.07457, 2019. [Online]. Available: http://arxiv.org/abs/1905.07457

[11] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 7762–7773.

[12] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" 2023.

[13] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, and C. Tian, "Enchanting program specification synthesis by large language models using static analysis and program verification," *arXiv preprint arXiv:2404.00762*, 2024.

[14] S. Chakraborty *et al.*, "Ranking llm-generated loop invariants for program verification," 2023.

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[16] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "Codegen2: Lessons for training llms on programming and natural languages," *arXiv preprint arXiv:2305.02309*, 2023.

[17] Y. Cao, S. Li, Y. Liu, Z. Yan, Y. Dai, P. S. Yu, and L. Sun, "A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt," *arXiv preprint arXiv:2303.04226*, 2023.

[18] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[19] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[20] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation policies from data. arxiv 2018," *arXiv preprint arXiv:1805.09501*, 1805.

[21] M. Bayer, M.-A. Kaufhold, and C. Reuter, "A survey on data augmentation for text classification," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–39, 2022.

[22] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 6023–6032.

[23] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.

[24] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 42–56. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908099

[25] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 42–56, 2016.

[26] M. Echenim, N. Peltier, and Y. Sellami, "Ilinva: Using abduction to generate loop invariants," in *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, ser. Lecture Notes in Computer Science, A. Herzig and A. Popescu, Eds., vol. 11715. Springer, 2019, pp. 77–93. [Online]. Available: https://doi.org/10.1007/978-3-030-29007-8_5

[27] J.-C. Filliâtre and A. Paskevich, "Why3—where programs meet provers," in *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer, 2013, pp. 125–128.

[28] M. Echenim, N. Peltier, and Y. Sellami, "A generic framework for implicate generation modulo theories," in *Automated Reasoning: 9th International Joint Conference, IJCAR 2018, Held as*

Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings 9. Springer, 2018, pp. 279–294.

[29] G. Kenison, L. Kovács, and A. Varonka, "From polynomial invariants to linear loops," *arXiv preprint arXiv:2302.06323*, 2023.

[30] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 499–512. [Online]. Available: https://doi.org/10.1145/2837614.2837664

[31] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[32] J. Laurent and A. Platzer, "Learning to find proofs and theorems by learning to refine search strategies: The case of loop invariant synthesis," *Advances in Neural Information Processing Systems*, vol. 35, pp. 4843–4856, 2022.

[33] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 106–120.

[34] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "Cln2inv: learning loop invariants with continuous logic networks," *arXiv preprint arXiv:1909.11542*, 2019.

[35] S.-W. Lin, J. Sun, H. Xiao, Y. Liu, D. Sanán, and H. Hansen, "Fib: Squeezing loop invariants by interpolation between forward/backward predicate transformers," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 793–803.

[36] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv:2109.00859*, 2021.

[37] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[38] OpenAI, "Gpt-4 technical report," 2023.

[39] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.