SLING: Using Dynamic Analysis to Infer Program Invariants in Separation Logic

Anonymous Author(s)

Abstract

We introduce a new dynamic analysis technique to discover invariants in separation logic for heap-manipulating programs. First, we use a debugger to obtain rich program execution traces at locations of interest on sample inputs. These traces consist of heap and stack information of variables that point to dynamically allocated data structures. Next, we iteratively analyze separate memory regions related to each pointer variable and search for a formula over predefined heap predicates in separation logic to model these regions. Finally, we combine the computed formulae into an invariant that describes the shape of explored memory regions.

We present SLING, a tool that implements these ideas to automatically generate invariants in separation logic at arbitrary locations in C programs, e.g., program pre/post conditions and loop invariant. Preliminary results on existing benchmarks show that SLING can efficiently generate correct and useful invariants for programs that manipulate a wide variety of complex data structures.

Keywords dynamic invariant generation, separation logic, program analysis

1 Introduction

A program invariant is a property that holds whenever program execution reaches a specific location. For example, a loop invariant can indicate a relation among the program variables at the loop entrance. Invariants can help prove program correctness, e.g., classical verification approaches by Floyd-Hoare and Dijkstra [13, 21] can be automated when given needed loop invariants and the infamous Heartbleed bug can be avoided by preserving an invariant capturing the proper size of the received payload message[16]. Invariants can also help developers understand programs, e.g., showing interesting or unexpected behaviors, and even discover non-functional bugs, e.g., revealing that the program has an unusual high runtime complexity [36]. Invariants are also useful in other programming tasks, including documentation, maintenance, code optimization, fault localization, program repair, and security analysis [1, 2, 15, 18, 30, 36, 44].

Unfortunately, software developers appear to perceive a "specification burden" [2] which leads them to eschew the writing of invariants in favor of executable code. For the past decade, researchers have been chipping away at this challenge of automatic invariant generation using static or dynamic analyses. A static analysis can reason about all program paths soundly, but doing so is expensive and is only possible to relatively small programs or simple forms of invariants, e.g., simple list structures [3, 11, 35]. Dynamic analysis focuses on program traces observed from running the program on small sample inputs, and thus provides no correctness guarantee on generated invariants. However, dynamic analysis is generally efficient and can infer expressive invariants because it only analyzes a finite, typically small, set of traces.

Existing invariant techniques often focus on invariants over scalar variables, e.g., relations among numerical values [19, 36, 41]. However, modern programs construct and manipulate data structures, i.e., highly-structured sets of memory locations within which these scalar values are stored. Example of such data structures are dynamically-allocated objects, e.g., heap-based objects created via the `new` keyword, standard data structures, e.g., lists and trees, or customized and user-defined structures that extend the standard ones and contain other structures internally. Understanding and reasoning about these heap-based programs are more challenging, e.g., even the task of accessing a variable requires checking if it points to a valid memory region (to avoid null pointer dereferencing).

An emerging approach to analyzing heap programs is to use invariants written in *separation logic* (SL) to represent memory structures [40, 46]. SL extends classical logic and allows for compact and precise representations of program semantics and reasoning to be localized to small portions of memory. In the last decade, research in SL has grown rapidly and led to practical techniques used in tools such as the Facebook Infer (FBInfer) analyzer [17].

Most existing SL works focus on static analyses to obtain sound results, and therefore can only consider simple classes of invariants or programs, e.g., to support the goal of "move fast to fix more things" [8, 39], FBInfer only considers simple data structures and restricts supported language features. Moreover, while many static analyzers, including FBInfer, compute SL invariants internally to verify programs, we are aware of only a few researchers who have investigated reifying those invariants for consumption by developers, and even then only for a restricted language of list manipulating [32] or tree traversing programs [5]. Also, most static SL tools aim to infer sufficiently strong invariants to achieve a specific goal, e.g., to prove memory safety or (programmer-provided) postcondition, and thus are not well suited for

PL'18, 2018, New York, NY, USA
2018.

discovering useful invariants to help understand code that lacks such formal specifications.

In this work, we introduce SLING (**S**eparation **L**ogic **In**variant **G**eneration), a tool that dynamically discovers SL invariants for heap programs. SLING takes as inputs a program, a location of interest, a set of predefined predicates defining data structures, and a set of sample inputs. SLING next runs the program on the inputs and uses a debugger to obtain traces capturing memory information of the variables at the considered location. These traces consist of the contents of the stack and heap of the program. SLING then iteratively analyzes variables using these traces to compute invariants. For each pointer variable, SLING generates SL formulas using predefined predicates to model the traces describing memory regions related to the variable. SLING also propagates computed information to improve the analysis of other variables in subsequent iterations. Finally, SLING combines the obtained formulae into a final invariant that represents the explored memory regions.

We use SLING to infer invariants for 157 C programs taken from two existing benchmarks [48] and [6]. These programs implement basic algorithms over standard data structures, e.g., singly-linked, doubly-linked, circular lists, binary trees, AVL, red-black trees, heaps, queues, stacks, iterators, etc, and complex functions from open source libraries and the Linux kernel that manipulate customized data structures. Preliminary results show that SLING can efficiently generate invariants that are correct and more precise than the documented specifications and invariants in these programs. Even when given incomplete traces, the tool can still discover partial invariants that are useful for users. We also show that SLING's invariants can help detect nontrivial bugs and reveal false positives in modern SL static analyzers such as FBInfer. We believe that SLING strikes a practical balance between correctness and expressive power, allowing it to discover complex, yet interesting and useful invariants out of the reach of the current state of the art.

2 Illustration

We describe SLING using the function concat shown in Figure 1. The function recursively concatenates two doubly linked lists x and y and returns (i) y when x is empty or (ii) a new doubly linked list by appending y to the tail of x. Although simple, concat requires several subtle preconditions over its inputs to work properly. First, y must be a nil-terminated list, i.e., the next field of its tail node is NULL, otherwise concat may not terminate when x contains a cycle or may refer to an unallocated memory region when the next field of x's tail node is a dangling pointer. Second, x and y must be non-overlapping, i.e., point to lists in separate memory regions, otherwise the resulting list contains a cycle.

```
typedef struct Node {
  struct node *next, *prev;
} Node;

Node *concat(Node *x, Node *y) {
  [L1]
  if (x == NULL) {
    [L2]
    return y;
  } else {
    Node *tmp = concat(x->next, y);
    x->next = tmp;
    if (tmp) tmp->prev = x;
    [L3]
    return x;
  }
}
```

Figure 1. Concatenating two doubly linked lists

As can be seen, such conditions can be difficult to analyze or even to specify because they involve dynamically-allocated data structures and their separations in memory. SLING aims to automatically discover such preconditions at program entrances and, more generally, invariant properties at arbitrary program locations, including postconditions and loop invariants.

2.1 Heap Predicates

SLING infers invariants expressed as formulae in separation logic (SL) to describe properties of heap-manipulating programs. Comparing to existing works for heap programs [26, 47], SL provides concise and expressive syntax and semantics to describe memory (shape) information [40, 46].

To analyze heap programs, SL works often use *inductive heap predicates* to compactly represent recursively-defined data structures. For concat, we use the predicate dll to define doubly linked lists:

$$\begin{array}{rcl}\mathsf{dll}(hd, pr, tl, nx) & \overset{\text{def}}{=} & (\mathsf{emp} \wedge hd{=}nx \wedge pr{=}tl) \\ & \vee & (\exists u.\ hd{\mapsto}u, pr * \mathsf{dll}(u, hd, tl, nx))\end{array}$$

The parameters hd, tl, pr, and nx point to the list's head, tail, previous, and next element, respectively. The definition of dll uses the built-in predicate emp to represent an empty heap, e.g., a NULL list, and the singleton predicate $x \overset{\mathsf{Node}}{\mapsto} nx, pr$ to denote a memory cell pointed to by a variable x of type Node shown in Figure 1 (nx and pr correspond to the next and prev fields of x, respectively[1]).

Conceptually, dll states that a doubly linked list is either an empty or a non-empty list, which is recursively defined by having the head hd point to a doubly linked list whose head node is u. In the latter case, the *separation conjunction*

[1]When the context is clear, we simply use $x{\mapsto}nx, pr$ for $x \overset{\mathsf{node}}{\mapsto} nx, pr$.

connector $*$ indicates the separation of memory regions modeled by *hd*'s singleton predicate and u's dll predicate, i.e., the heaplets of *hd* and u are disjoint.

SLING uses such heap predicates to discover invariants and specifications of heap programs. For concat, SLING uses dll to generate the precondition on line 6 and the postcondition on line 14 in Figure 1 as

$$\begin{aligned} \texttt{pre} &= \exists p,u,v.\ \mathsf{dll}(\mathtt{x},p,u,\mathsf{nil}) * \mathsf{dll}(\mathtt{y},\mathsf{nil},v,\mathsf{nil}) \\ \texttt{post} &= \exists v.\ \mathsf{dll}(\mathtt{y},\mathsf{nil},v,\mathsf{nil}) \wedge \mathtt{x}{=}\mathsf{nil} \wedge \mathtt{res}{=}\mathtt{y} \vee \\ &\quad\ \exists p,u,v.\ \mathsf{dll}(\mathtt{x},p,u,\mathtt{y}) * \mathsf{dll}(\mathtt{y},u,v,\mathsf{nil}) \wedge \mathtt{res}{=}\mathtt{x} \end{aligned}$$

These pre and postconditions form a valid specification for concat. The precondition requires that the inputs x and y point to two disjoint nil-terminated doubly linked lists. Note that unlike y, the dll of x shows that it can take any arbitrary previous pointer, i.e., the existential argument p, because this pointer changes across the recursive call on line 11.

The postcondition ensures two exit conditions: (i) when x is empty, the return value res is y, and (ii) otherwise, res = x is the result of appending y to x by changing the next element of x's predicate from nil to y. Also, note that the previous field of y now points to the tail element of x. Lastly, the postcondition states that the separation of the heaps of x and y is preserved, i.e., concat only changes the field values of the lists and does not alter the allocated memory.

Although inductive heap predicates such as dll appear complex, they compactly capture crucial shape properties e.g., doubly linked lists are acyclic, and are standard in SL, e.g., provided by the users or predefined in an analyzer [7, 24, 29, 33]. In addition, compared to normal, non-SL predicates such as isOdd or $x \geq y$, checking SL heap predicates is nontrivial because we have to "unfold" data structures recursively and find concrete values to instantiate the existential quantifers. Thus, inferring such SL predicates is not simply checking a given predicate against given traces, e.g., the pre/post conditions above require finding the correct quantified variables and parameters in dll.

2.2 Traces

Given a program annotated with locations of interest, SLING runs the program on sample inputs to collect execution traces. Currently, we use the LLDB debugger [31] to observe execution traces containing memory addresses and values of the variables in scope at considered locations[2].

Figure 2a shows two doubly linked lists x and y of size 3 and 2, respectively. When run on these inputs, SLING records traces such as those given in Figures 2. Figure 2a shows the trace obtained in the first iteration of concat at $L1$. The trace contains information about both the *stack*, consisting values of the variables accessible at this location, e.g., x=0x01, y=0x04, and the *heap*, consisting allocated memory cells reachable from the stack's variables, e.g., 0x01↦node{next:0x02;prev:nil}.

[2]These "traces" are often referred to as the *stack-heap models* in SL literature, which we review in Section 3.

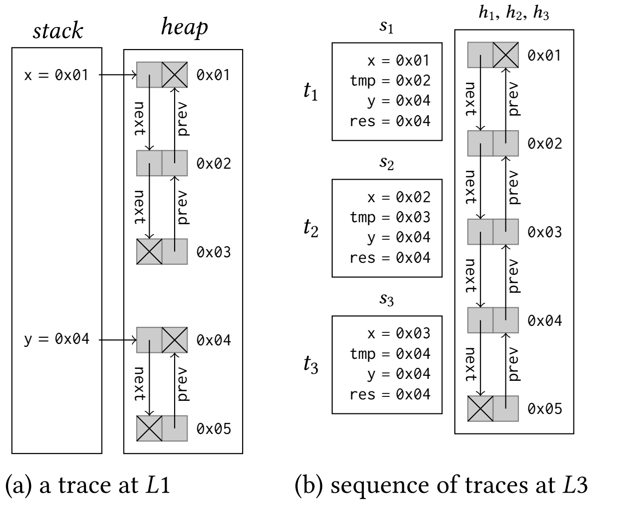

Figure 2. Traces collected at $L1$ and $L3$ in concat.

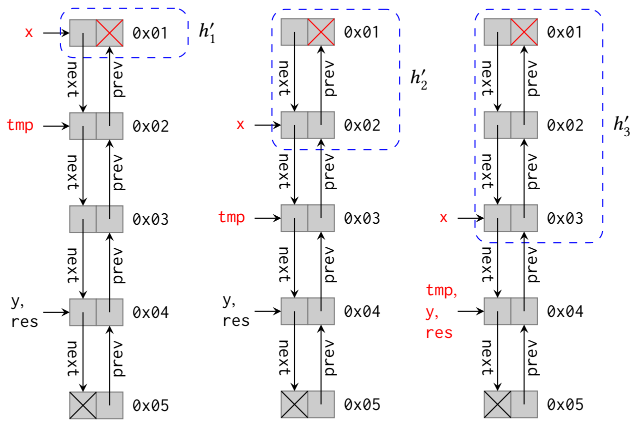

Figure 3. Sub-heaps of x (in blue dashed boxes) and their boundaries (nil and the variables in red). The common boundary of these sub-heaps is {x, tmp, nil}.

Figure 2b shows three set of traces collected at $L3$ for the first three iterations of concat. Note that the values of x and tmp are different in the stacks because x and tmp change across the recursive calls in concat. However, the heap is similar in the traces because concat does not change the heap, e.g., delete or create cells, and all memory cells are still reachable from the stack variables. Moreover, the stack at $L3$ also contains a *ghost* variable res which stores the function's return value[3].

2.3 Inference

SLING infers invariants consisting of SL predicates such as dll over variables at a location of interest. For each (pointer) variable, SLING explores relevant memory regions in observed traces to compute invariant properties of that variable. Finally, SLING combines the computed invariants to model the whole explored memory regions.

Postcondition. We demonstrate how SLING computes the postcondition at $L3$ using the predicate dll and the traces shown in Figure 2b. We assume that SLING analyzes the variables at $L3$ in the order x, tmp, y, res and discuss the effect of different orders in a moment.

SLING first analyzes the traces to compute the sub-heaps of x and their boundaries. The *sub-heaps* contain memory cells reachable from x but not pointed to by other stack variables. The *boundaries* of x's sub-heaps contain x itself, the nil pointer if it is reachable from x, and all variables reachable by x or its aliasing pointers. Next, SLING takes the intersection of the boundaries to obtain the *common boundary*, which consists of variables that we use to compute invariants in the next step. In our running example, as shown in Figure 3, the computed sub-heaps $h'_1 = \{\texttt{0x01}\}$, $h'_2 = \{\texttt{0x01}, \texttt{0x02}\}$, and $h'_3 = \{\texttt{0x01}, \texttt{0x02}, \texttt{0x03}\}$ over the three traces of x and their respective boundaries $\{\mathtt{x}, \mathsf{nil}, \mathtt{tmp}\}$, $\{\mathtt{x}, \mathsf{nil}, \mathtt{tmp}\}$, and $\{\mathtt{x}, \mathsf{nil}, \mathtt{tmp}, \mathtt{res}, \mathtt{y}\}$. Their common boundary is then $\{\mathtt{x}, \mathsf{nil}, \mathtt{tmp}\}$.

From the computed sub-heaps and boundary variables, SLING searches the predefined predicates for formulae that are consistent with the sub-heap traces using the boundary variables. For each predicate, SLING creates *candidate* formulae by instantiating the predicate parameters with the boundary variables. It does so by enumerating different subsets of the boundary variables as predicate parameters. For subsets of size smaller than the number of parameters, SLING introduces fresh existential variables to instantiate the predicate. In our example, SLING enumerates formulae such as $\exists u_1.\, \mathsf{dll}(\mathtt{x}, \mathsf{nil}, \mathtt{tmp}, u_1)$, $\exists u_1.\, \mathsf{dll}(\mathtt{x}, \mathsf{nil}, u_1, \mathtt{tmp})$, ...

Next, SLING uses an SMT-based model checker to check each candidate formula against the given sub-heaps. The checker either refutes the formula, which is then discarded, or accepts it, which SLING then considers as a valid formula over the sub-heaps. Intuitively, accepted formulae represent partial invariants computed from memory regions, e.g., the sub-heaps, related to the analyzed variable.

In our example, among the generated candidates, the checker accepts the formula $F_x = \exists u_1, u_2.\, \mathsf{dll}(\mathtt{x}, u_1, u_2, \mathtt{tmp})$. This formula shows that x is a doubly linked list to the *next* pointer tmp. The existential variables u_1, u_2 indicate that SLING cannot find concrete stack or nil variables for the second and third parameters of dll from the traces.

[3]This value is captured when the LLDB debugger steps out of the function and jumps back to its call site.

Although F_x holds over the given sub-heaps, it might not generalize to the entire heap in the observed traces. Thus, when analyzing F_x, the checker also computes a *residual heap*, which represents the part of the heap that is not modeled by F_x. The checker also computes a mapping from existential variables to concrete memory addresses from given traces. SLING propagates the residual heap and address mapping to improve the analyses of other variables in subsequent iterations.

SLING now continues with the other variables tmp, y, res using the described steps and computed information (residual heaps and address mappings). For tmp, SLING computes the sub-heaps and boundary $\{\mathtt{tmp}, \mathtt{x}, \mathtt{y}, \mathtt{res}\}$, and then computes the formula $F_{\mathtt{tmp}} = \exists u_3.\, \mathsf{dll}(\mathtt{tmp}, \mathtt{x}, u_3, \mathtt{y})$, indicating a doubly linked list from tmp to the next pointer y. Also, observe that the previous pointer points to x, showing the connection between this list and the one modeled by F_x.

Similarly, for the last two variables y and res, SLING obtains the formulae $F_y = \exists u_4, u_5.\, \mathsf{dll}(\mathtt{y}, u_4, u_5, \mathsf{nil})$ and $F_{\mathtt{res}} = \mathsf{emp}$. $F_{\mathtt{res}}$ is emp because every sub-heap reachable from res is empty in the traces observed in the last iteration of the program.

The obtained atomic formulae $F_{\mathtt{x}}, F_{\mathtt{y}}, F_{\mathtt{tmp}}$, and $F_{\mathtt{res}}$ model separate sub-heaps, thus SLING combines them using the $*$ operator to form a *shape* invariant capturing the shape of the memory at $L3$, e.g., connections among separate heaplets:

$$F_{L3} = \exists u_1, u_2, u_3, u_4, u_5, \mathtt{tmp}.$$
$$\mathsf{dll}(\mathtt{x}, u_1, u_2, \mathtt{tmp}) * \mathsf{dll}(\mathtt{tmp}, \mathtt{x}, u_3, \mathtt{y}) * \mathsf{dll}(\mathtt{y}, u_4, u_5, \mathsf{nil}).$$

Note that the weakest constraint $F_{\mathtt{res}} = \mathsf{emp}$ is discarded from the conjunction. Also note that the local variable tmp is not in the scope of the function's exit, thus SLING considers it as an existential variable in F_{L3}. In general, SLING only uses the function's parameters and the ghost variable res as free variables in the function's pre and postconditions.

SLING also examines all analyzed information to find additional *pure* (not related to memory) relations among the stack and existential variables in the inferred formula. In this example, SLING determines that $\mathtt{res} = \mathtt{x}$, indicating that the return value at $L3$ is x. It also infers aliasing information such as $\mathtt{x} = u_2$, $u_3 = u_4$ from the address mapping.

From these additional equalities, SLING derives the final result:

$$F'_{L3} = \exists u_1, u_3, u_5, \mathtt{tmp}.\, \mathsf{dll}(\mathtt{x}, u_1, \mathtt{x}, \mathtt{tmp}) *$$
$$\mathsf{dll}(\mathtt{tmp}, \mathtt{x}, u_3, \mathtt{y}) \; * \; \mathsf{dll}(\mathtt{y}, u_3, u_5, \mathsf{nil}) \wedge \mathtt{res} = \mathtt{x}.$$

This result F'_{L3} is correct at $L3$ and even more precise than the postcondition shown in Section 2.1 when $\mathtt{x} \neq \mathsf{nil}$: $\exists p, u, v.\, \mathsf{dll}(\mathtt{x}, p, u, \mathtt{y}) * \mathsf{dll}(\mathtt{y}, u, v, \mathsf{nil}) \wedge \mathtt{res} = \mathtt{x}$. The reason is because $\mathsf{dll}(\mathtt{x}, u_1, \mathtt{x}, \mathtt{tmp}) * \mathsf{dll}(\mathtt{tmp}, \mathtt{x}, u_3, \mathtt{y})$ in F'_{L3} entails $\exists p, u.\, \mathsf{dll}(\mathtt{x}, p, u, \mathtt{y})$ in the given postcondition. The reversed direction of this entailment does not hold as it requires a non-trivial condition $\mathtt{x} \neq \mathtt{y}$.

Precondition and Other Invariants. Using the same inference process over the traces obtained from the input x, y given in Figure 2, SLING infers the precondition at location $L1$ and the invariant at location $L2$ of `concat` as

$$\begin{aligned} F'_{L1} &= \exists u_1, u_2, u_3, u_4. \\ &\qquad \mathsf{dll}(\mathtt{x}, u_1, u_2, \mathsf{nil}) * \mathsf{dll}(\mathtt{y}, u_3, u_4, \mathsf{nil}) \land u_3 = \mathsf{nil}, \text{ and} \\ F'_{L2} &= \exists u_1, u_2. \\ &\qquad \mathsf{dll}(\mathtt{y}, u_1, u_2, \mathsf{nil}) \land u_1 = \mathsf{nil} \land \mathtt{x} = \mathsf{nil} \land \mathtt{res} = \mathtt{y}. \end{aligned}$$

These are the pre and postconditions shown in Section 2.1. From these results, we obtain the specification of `concat` because the complete post-condition is disjunction $F'_{L2} \lor F'_{L3}$. In general, SLING can compute invariants at arbitrary program locations by applying the described inference process to traces obtained at those locations.

Note that, depending on the program, the order of the analyzed variables can affect the resulting invariants, i.e., make them stronger or weaker. This is because the propagated residue information affects the computation of boundary variables and thus affects the instantiations of parameters in the predicates. However, in our experiment (Section 5.3), we use SLING with a fixed order and obtain results that are comparable and often more precise than the expected invariants.

3 Separation Logic

SL has been actively used to recently to reason about imperative programs that manipulate data structures. Crucially, SL uses the separating conjunction operation $*$ to describe the separation of computer memory. More specifically, the assertion $p * q$ states that the memory portion of which can be decomposed into two *disjoint* sub-portions held by p and q respectively. In addition, SL is often equipped with the ability for users to define inductive heap predicates (such as the predicate dll for doubly linked list used in Section 2). The combination of the $*$ operator and predicates make SL expressive enough to model various types of data structures, such as lists and trees.

Figure 4 shows the syntax and semantics of the SL formulas we consider in this work. These represent the standard symbolic-heap fragment of SL [7] with user-defined inductive heap predicates.

Syntax. We denote x as a variable, k, e as an integer constant and an integer expression, respectively, nil as a constant denoting a dangling memory address (*null*), and a as a spatial expression modeling a memory address. The predicate emp models an empty heap, the singleton heap predicate $x \overset{\tau}{\mapsto} t_1, ..., t_n$ models an n-field data structure type τ where x points to, and the inductive heap predicate $\mathsf{P}(t_1, ..., t_n)$ models a recursively defined data structure. The *spatial* formulas Σ consist of these predicates and their compositions using the $*$ separating conjunction operator. Π denotes *pure formulas* in linear arithmetic, which do not contain any predicates. Note that we can negate the presented formulas to obtain formulas involving disjunctions, universal quantifiers, and other comparison relations.

Semantics. Given a set `Var` of variables, `Type` of types, `Val` of values, and `Loc` $\subset$ `Val` of memory addresses, an SL *stack-heap model*, i.e., concrete trace, is a pair of a *stack model* s, which is a function s: `Var` $\rightarrow$ `Val`, and a *heap model* h, which is a partial function h: `Loc` $\rightharpoonup$ (`Type` $\times$ `Val`*).

We write $[\![\Pi]\!]_s$ to denote the valuation of a formula Π under the stack model s and $s, h \models F$ to denote a model s, h satisfies a formula F. Moreover, dom(h) denotes the domain of h, $h_1 \# h_2$ denotes h_1 and h_2 have different domains, and $h_1 \circ h_2$ denotes the union of two disjoint heaps h_1 and h_2, and $[f \mid x : y]$ indicates a function like f except that it returns y for input x.

Model Checking. We follow the technique given in [7] to implement a model checker, which checks if a formula F is satisfied by a stack-heap model s, h and returns a residual heap h', i.e., memory regions in h not modeled by F, and an instantiation ι that maps existential variables in F to concrete addresses in the model. These checking and instantiation tasks are encoded as logical formulas solvable using the Z3 SMT solver [12].

Note that the model checking technique proposed in [7] does not return the instantiation ι, which is needed by SLING to compute equalities among variables in F. To obtain ι, we slightly redefine the problem with a new satisfaction relation:

Definition 1 (Satisfaction Relation with Existential Instantiation)**.** *The relation $s, h \models_\iota F$ is the satisfaction relation $s, h \models F$ except that the value of an existential variable in F is obtained from the instantiation ι, which is a function from* `Var` *to* `Val` *similar to the stack model.*

Definition 2 (Symbolic-heap Model Checking)**.** *A reduction $s, h \Vdash F \leadsto h', \iota$ is valid if $h' \subseteq h$ and $s, h \setminus h' \models_\iota F$.*

Definition 2 redefines the model checking reduction relation to return, in addition to the residual heap model h', an instantiation ι of existential variables in F such that the satisfaction relation $s, h \setminus h' \models_\iota F$ defined in Definition 1 holds.

4 The SLING Algorithm

Figure 5 shows the main algorithm of SLING. SLING takes as input a program C, a set P of pre-defined inductive heap predicates, a target location l in C, and a test suite T, and returns a set R of SL formulas that are satisfied by all collected traces at l.

SLING infers invariants using the three main phases described below. In the following we use the term *stack-heap models* to refer to concrete traces.

Syntax				**Semantics**		
e	::=	$k \mid x \mid -e \mid e_1 + e_2 \mid k \cdot e$	Integer exps	$s, h \models \mathsf{emp}$	iff	$\mathrm{dom}(h) = \varnothing$
a	::=	$\mathsf{nil} \mid x$	Spatial exps	$s, h \models x \overset{\tau}{\mapsto} t_1, \ldots, t_n$	iff	$\mathrm{dom}(h) = \{s(x)\}$ and $h(s(x)) = (\tau, \{s(t_1), \ldots, s(t_n)\})$
Π	::=	$a_1 = a_2 \mid e_1 = e_2 \mid e_1 < e_2 \mid$	Pure formulas	$s, h \models \mathsf{P}(t_1, \ldots, t_n)$	iff	$s, h \models F$, where $F \overset{\text{def}}{\Rightarrow} \mathsf{P}(t_1, \ldots, t_n)$
		$\neg\Pi \mid \Pi_1 \wedge \Pi_2 \mid \exists x.\, \Pi$		$s, h \models \Sigma_1 * \Sigma_2$	iff	$\exists h_1, h_2.\, h_1 \# h_2 \wedge h_1 \circ h_2 = h \wedge s, h_1 \models \Sigma_1 \wedge s, h_2 \models \Sigma_2$
Σ	::=	$\mathsf{emp} \mid x \overset{\tau}{\mapsto} t_1, \ldots, t_n \mid$	Spatial formulas	$s, h \models \Sigma \wedge \Pi$	iff	$[\![\Pi]\!]_s = \mathsf{true}$ and $s, h \models \Sigma$
		$\mathsf{P}(t_1, \ldots, t_n) \mid \Sigma_1 * \Sigma_2$		$s, h \models \exists x.\, F$	iff	$\exists v \in \mathtt{Val}.\, [s \mid x : v], h \models F$
F	::=	$\Sigma \mid \Pi \mid \Sigma \wedge \Pi \mid \exists x.\, F$	SL formulas			

Figure 4. Syntax and Semantics of Symbolic-heap SL formulas.

```
Procedure SLING(C, P, T, l)
Input: A program C, a set of pre-defined predicates P, a test suite T, and a program location l
Output: A set of invariants at the location l
 1: SH ← CollectModels(C, l, t)
 2: V ← GetVars(l)                                         ▹ stack variables
 3: R ← {(emp, SH, $(\iota_i = \{\})_t$)}                    ▹ initial result
 4: for each pointer $v$ in V do
 5:     R′ ← {}
 6:     for each ($F$, SH, I) in R do
 7:         SH$_v$, SH$_r$, B ← SplitHeap(SH, $v$)   ▹ SH$_v$ ⊕ SH$_r$ ≡ SH
 8:         R$_v$ ← InferAtom($v$, SH$_v$, B, P)
 9:         for each ($F_v$, SH′, I′) in R$_v$ do
10:             R′ ← R′ ∪ {($F * F_v$, SH$_r$ ⊕ SH′, I ⊕ I′)}
11:     R ← {(InferPure($F$, SH, I), SH, I) | ($F$, SH, I) ∈ R′}
12: return R
```

Figure 5. Main algorithm

Model Collection (line 1). SLING uses the procedure CollectModels to collect all stack-heap models observed at the target location l when running the program over the tests in T. In particular, CollectModels calls a software debugger such as LLDB to set a breakpoint at l and inspect the memory layout when executing the program. It then collects the set of stack-heap models SH from the memory whenever hitting the breakpoint at l.

Inference (lines 2–11). After obtaining stack-heap models at l, SLING performs a *heap* inference and then a *pure* inference to derive a result set satisfied by all stack-heap models. SLING uses an iterative refinement process over the stack variables to infer invariants. At each iteration, SLING updates the result set R with a tuple (F, SH, I), where the formula F holds for the models analyzed in the previous iteration, SH captures the residue of the initial heaps that are not modeled by F, and the sequence I contains existential instantiations which map the existential variables in F to concrete memory addresses.

In each iteration, given a stack variable $v \in$ V and a tuple (F, SH, I) $\in$ R, SLING derives a set of atomic heap predicates, i.e., inductive heap predicates, singleton heap predicates, or emp, modeling sub-heaps in SH that contain memory cells reachable from v. By the loop refinement's invariant, the heaps modeled by these predicates and F are disjoint, so that we can strengthen F with each of them via the $*$ operator of SL. Intuitively, SLING splits the original stack-heap models into multiple sub-heaps, which are pointed-to by distinct (non-aliasing) stack variables. To model a sub-heap, SLING derives atomic formulas from the given set of predicates and the stack variables related to the sub-heap. Two subprocedures SplitHeap and InferAtom will be elaborated in Sections 4.1 and 4.2, respectively.

In addition to finding invariants describing shape properties, SLING infers equality constraints over stack variables in V to represent pure properties. This step is captured on by the InferPure procedure on line 11 and is discussed in Section 4.3.

Validation. When we infer both pre and post-conditions, we can combine them to obtain program specifications. In addition, we leverage the *frame rule* of SL to check if this combination is consistent with respect to corresponding residual models. Thus, when inferring invariants at multiple `return` statements, SLING has an additional phase that combines and validates formulae inferred at different locations to eliminate spurious details from the final results. We will elaborate this validation in Section 4.4.

4.1 Heap Partitioning

Given a sequence SH of collected stack-heap models, SLING calls SplitHeap to splits the heap in each model $s_i, h_i \in$ SH into smaller sub-heaps so that each of them can be modeled by atomic heap predicates. Moreover, SplitHeap returns the common boundary of these sub-heaps, which consist of the nil and stack variables that are subsequently used to determine the arguments for these atopic heap predicates.

SplitHeap uses depth-first search to traverse the pointer fields of memory cells from a *root* pointer to partition the heap model h_i into two non-overlapping parts:

Procedure InferAtom($root$, SH_{root}, B, P)

Input: A stack pointer $root$, its sub-models SH_{root} and their common boundary B, and a set of predefined predicates P
Output: A set of atomic formulas modeling the sub-models and their residue information

```
1:  $R \leftarrow \{\}$
2:  for each $P(t_1, ..., t_n)$ in P do                         ▹ Consider a predicate P
3:      $\widehat{A} \leftarrow \{A \mid A \in \mathrm{PowerSet}(B) \land |A| \le n \land root \in A_i\}$
4:      for each A in $\widehat{A}$ do                          ▹ Consider a subset of B
5:          $\{u_1, ..., u_m\}_{m=n-|A|} \leftarrow \mathrm{fresh}(n-|A|)$
6:          $A \leftarrow A \cup \{u_1, ..., u_m\}_{m=n-|A|}$
7:          for each permutation $(k_1, ..., k_n)$ in $\mathrm{Perm}_n(A)$ do
8:              if $\forall 1 \le i \le n.\, k_i \in B \rightarrow \mathrm{type}(k_i) <: \mathrm{type}(t_i)$ then
9:                  $F \leftarrow \exists u_1, ..., u_m.\, P(k_1, ..., k_n)$
10:                 if $\forall s_i, h_i \in \mathsf{SH}_{root}.\, s_i, h_i \Vdash F \rightsquigarrow h_i', \iota_i$ then
11:                     $R \leftarrow R \cup \{(F, ((s_i, h_i'))_i, (\iota_i)_i)\}$
12: if $\forall s_i, h_i \in \mathsf{SH}_{root}.\, |h_i| = 1$ then
13:     $R \leftarrow R \cup \mathrm{InferSingleton}(root, \mathsf{SH}_{root})$
14: if $R = \emptyset$ then $R \leftarrow \{(\mathrm{emp}, \mathsf{SH}_{root}, (\iota_i = \{\})_i)\}$
15: return R
```

Figure 6. Inferring Atomic Predicates

- The sub-heap h_i' containing all memory cells in h_i reachable from the $root$ pointer variable up to the nil pointer or memory cells pointed-to by other stack pointer variables. We call these pointer variables or the nil pointer the *boundary* between the sub-heap h_i' and the other memory regions in the heap h_i.
- The remaining sub-heap $h_i'' = h_i \setminus h_i'$. The sub-heap h_i'' may contain memory cells unreachable from $root$ and those reachable from $root$, but also pointed-to by other stack variables non-aliasing with $root$.

Figure 3 illustrates the boundaries of the sub-heaps h_1', h_2', and h_3' of the root variable x in concat example. The respective boundaries of h_1', h_2' are {x, nil, tmp}, {x, nil, tmp}, and {x, nil, tmp, res, y}. Note that although the heaps in the original models are the same, their stacks are different, which result in different sub-heaps and different boundaries. Their common boundary is {x, nil, tmp}.

4.2 Inferring Atomic Heap Predicates

Given the sequence of sub-models SH_{root} of the $root$ pointer and its boundary B, the procedure InferAtom in Figure 6 computes a set of atomic predicates such that these predicates are satisfied by all sub-models in SH_{root}. These atomic (shape) predicates consist of either (i) inductive heap predicates whose definitions are given in the set P (lines 2–11), (ii) singleton predicates of the $root$ pointer when the heap size of all sub-models in SH_{root} is 1 (lines 12–13), or (iii) the emp predicate with SH_{root} as the residual models when it cannot derive any predicates in the two former forms (line 14).

Inductive Heap Predicates. SLING discovers instances of each predicate $P(t_1, ..., t_n)$ which is pre-defined in P. For optimization purpose, the set P of predicate definitions may be filtered to contain only predicates which have at least one parameter being the same type as the $root$ pointer. Also, for simplicity of presentation, we assume that all n parameters $t_1, ..., t_n$ of the predicate P are the pointer types.

SLING chooses potential arguments of the predicate P from the common boundary B of all sub-heaps in the sub-models SH_{root}. It searches for these arguments from all permutation of B's subsets whose size is less than or equal to n and contains $root$ (line 3). We expect the inferred inductive predicates to contain as many stack variables as its arguments as possible, therefore we examine each subset in the ascending order of their size. Also, to reduce the search space, we only consider a permutation $(k_1, ..., k_n)$ if it is type-consistent with the parameters $t_1, ..., t_n$ of the predicate P; that is, if k_i is a stack pointer variable then its type must be a subtype of the corresponding parameter t_i's type (line 8). Then, we construct a formula F from the inductive heap predicate $P(k_1, ..., k_n)$ (line 9). A formula F is valid if it is successfully checked by all models in SH_{root} (line 10). This validity check also returns a residual heap h_i' and an existential instantiation ι_i for each stack-heap model s_i, h_i in SH_{root}. They are respectively the member of the sequence of residual models and the sequence of existential instantiations associating with the valid formula F as an inference result in the set R (line 11).

In the concat example, when selecting the argument set {x, tmp, nil} ∈ A, the algorithm derives the formula $F_x = \exists u_1.\, \mathrm{dll}(u_1, \mathrm{nil}, \mathrm{x}, \mathrm{tmp})$. This result shows that x is the last node of doubly linked lists whose head is u_1 and its next pointer is tmp. Moreover, the formula F_x models the whole sub-heaps of x in Figure 3, i.e., all residue models have empty heaps, when the existential variable u_1 is instantiated to the address 0x01. As another example, when considering the another set of potential arguments {x, tmp} ∈ A, we infer $F_x = \exists u_1, u_2.\, \mathrm{dll}(\mathrm{x}, u_1, u_2, \mathrm{tmp})$, indicating that x is the head of a doubly linked list segment to tmp.

Singleton Heap Predicates. We only derive singleton heap predicates of $root$ when there is a single memory cell in *every* $root$'s sub-model in SH_{root} (line 12). We consider a τ-typed singleton predicate template of the form $root \overset{\tau}{\mapsto} k_1, ..., k_n$. The value of each field k_i in the template is the common pointer variable (including nil) pointing to the corresponding field of every memory cell in SH_{root}. If there is no such variable, we create a fresh existential variable for k_i and update this variable's instantiation to the value of the corresponding field in each model.

4.3 Pure Inference

The heap predicates derived in the previous steps mainly present the heap memory, but not the relations of variables within a predicate and among predicates in the overall results. In these results, the heap predicates are solely related via the common stack variables in their arguments.

We infer additional pure constraints over arguments of the predicates by searching for equality constraints over two different variables among stack variables, existential variable, `nil`, and the special variable `res` if we are inferring post-conditions which are satisfied by every stack model and existential instantiation.

For example, we use this inference to obtain the relation $\mathtt{res} = \mathtt{y}$ about the return value of `concat` and other aliasing information, e.g., those shown in Section 2.3.

4.4 Validation

When we obtain multiple postconditions (e.g., at each `return` statements), we can combine them with the inferred preconditions to obtain program specifications. In addition, we can exploit the frame rule of separation logic to validate the resulting specification.

Consider the frame rule of separation logic

$$\frac{\{P\}\mathsf{C}\{Q\}}{\{R * P\}\mathsf{C}\{R * Q\}}$$

The rule says that if a triple $\{P\}\mathsf{C}\{Q\}$ is valid (i.e., C executes safely in the precondition P and its post-states satisfying the postcondition Q) then the triple $\{R * P\}\mathsf{C}\{R * Q\}$, in which R is an frame axiom modeling memory regions that are not manipulated by C, also holds.

Assume that P and Q are the precondition and postcondition of a command C derived by our inference algorithms. In our settings of dynamic analysis, if C is a method, then these assertions are inferred from the stack-heap models observed at the entry and exit of C, respectively. On the other hand, if C is a loop body, then P and Q are identical and considered as a loop invariant. This invariant is inferred from the stack-heap models collected at the start of the loop's iterations.

According to the frame rule, if the inferred precondition P and postcondition Q is valid, then we can expand the corresponding memory regions modeled by P and Q by the *same* memory regions non-overlapping with them to become the whole memories observed at the entry and exit C. Otherwise, the inference result (P, Q) is considered as spurious. Therefore, we can check that the residual models corresponding to P and Q are *unchanged* with respect to all observed models to determine if this result is spurious or not.

For the `concat` example in Section 2, we obtain a precondition F'_{L1} and two postconditions F'_{L2} (when $\mathsf{x} = \mathsf{nil}$) and F'_{L3} (when $\mathsf{x} \neq \mathsf{nil}$). For each pair of a model collected at $L1$ and the corresponding model collected at $L2$ or $L3$, we check if the residual heap in the model at $L1$, which is not captured by F'_{L1}, is the same as the residual heap in model at $L2$ or $L3$, which is not captured by F'_{L2} or F'_{L3}, respectively. For example, given the model at $L1$ in Figure 2a and its corresponding model t_1 at $L3$ in Figure 2b, the residual heaps corresponding to F'_{L1} and F'_{L3} are both empty. On the other hand, in the last iteration of `concat` (when $\mathsf{x} = \mathsf{nil}$), the residual heaps of F'_{L1} and F'_{L2} both contain three memory cells `0x01`, `0x02`, and `0x03`.

4.5 Complexity

SLING is exponential in the number of predicates and their parameters. In addition, the complexity of the general heap model checking problem is EXPTIME [7]. Therefore, checking predicates over combinations of variables over many collected stack-heap trace models can be slow. Currently, SLING uses a type-checker (Figure 6, line 8) to eliminate variable combinations having inconsistent types to observed stack-heap models. Our experiments in Section 5 show that only a few traces are needed to discover accurate and useful invariants. Moreover, despite the EXPTIME complexity, the Z3-based model checker performs efficiently in our experiments.

5 Evaluation

SLING is implemented in Python and uses the LLDB debugger [31] to collect traces at target program locations. Below we evaluate SLING on C programs, but SLING also works with programs written in other languages supported by LLDB (e.g., C++ and Objective-C) or having debuggers capable of capturing memory information (e.g., JDB for Java [25], PDB for Python [42], and GDB [20]).

Our experiments described below are conducted on a computer with 2.2GHz Intel CPU, 16 GB memory, and runs Mac OS. The source code of SLING and experimental data are available at https://bitbucket.org/anonymized/sling/.

5.1 Benchmark Programs

We evaluate SLING using the VCDryad benchmark [48] consisting 153 C heap-manipulating programs collected from various verification works, e.g., SV-COMP [4], GRASShopper [45] and AFWP [23]. These programs range from those that manipulate standard data structures (e.g., heaps and trees) to functions from popular open source libraries (e.g., Glib, OpenBSD) and the Linux kernel that manipulate customized data structures. Some of these programs have non-trivial bugs (e.g., causing segmentation faults) intended to test static analyzers. We also use 4 programs[4] from [7]. These programs implement non-trivial algorithms using multiple data structures (e.g., the Schorr-Waite graph marking algorithm using binary trees).

[4]This benchmark has 6 programs, we use 4 of them and exclude the other 2 because they use concurrency which SLING currently does not support.

In total, these benchmark programs contain a wide variety of structures including singly-linked lists, doubly-linked lists, sorted lists, circular lists, binary trees, AVL trees, red-black trees, heaps, queues, stacks, iterators, etc. Moreover, these programs contain documented invariants (e.g., pre/postconditions such as those given in Section 2.1), which we use to evaluate SLING's inferred invariants.

Table 1 shows these 157 programs (the last row shows the 4 programs from [7]). Column **Programs** lists the programs, categorized by data structures that they use. Column **LoC** shows the total line of code of these programs. For example, the first row lists 8 programs that use standard singly-link lists (SLL) and have in total 168 LoC. In total, we have 157 programs in 22 categories with 4649 lines of C code.

5.2 Setup

For each program, we obtain traces, i.e., stack-heap models, to infer invariants at program entrances for preconditions, at loop entrances for loop invariants, and at program exits for postconditions (we systematically obtain traces at each return statement in a program). We use LLDB to set breakpoints at these locations to collect traces.

To obtain traces, we run each program on empty and randomly generated data structure inputs of a fixed size of 10. For example, for the `concat` program in Figure 1 that takes as input 2 doubly-linked lists, we generate 3 inputs consisting of a nil list and two randomly generated doubly-linked lists a, b of size 10 and run `concat` over all combinations over these inputs, e.g., `(nil,a)`, `(nil,b)`, `(a,b)`, ... Although these inputs are random and relatively small (size 10), the benchmark programs often modify and loop over data (e.g., as in `concat`), allowing us to generate sufficient and diverse traces.

For each data structure category shown in Table 1, we adopt the predicate definitions given for that data from the benchmark programs, e.g., all programs in the **DLL** category use the dll inductive predicate shown in Section 2. The shape and complexity of these predicates vary, e.g., dll has 4 parameters, 1 singleton predicate, and 1 inductive predicate, and the `treeSeg` predicate has 2 parameters, 2 singletons, and 4 inductive predicates.

Programs in several categories such as `SV-COMP` and `Cyclist` use complex *nested* data structures, which are data structures whose fields are other data structures. The predicates for these data structures involve multiple predicates that are quite complex, e.g., the `iter` predicate has 10 parameters, 5 singleton and 6 inductive predicates.

5.3 Results

Table 1 shows our results. Columns **iLocs**, **Traces**, and **Invs** lists, for the programs in the category, the total number of target locations, number of obtained traces, and number of generated invariants, respectively. The **Invs** column also lists the number of spurious invariants in parentheses (rows with no such parentheses have no spurious invariants). Finally, column **Time** lists the total analysis time in seconds (including program execution, trace collection, and invariant inference).

For several programs, we were not able to obtain traces at considered locations using random inputs and thus could not infer invariants at those locations. Column **A/S/X** shows the number of programs where we (A) obtained traces at all considered locations, (S) obtained traces for some locations or inferred spurious results (X) could not obtain traces or infer invariants (timeout) at some locations. For example, for the 5 programs using binary search trees, we obtained traces at all considered locations in 2 program, obtained traces at some locations in 2 programs, and could not obtain any traces in one program (`quicksort`).

The last three columns in the table give additional details about the generated invariants. Columns **Single**, **Pred**, and **Pure** list the average numbers of singleton predicates (e.g., $x \overset{\text{node}}{\mapsto} nx, pr$), inductive predicates (e.g., dll), and pure equalities (e.g., $x =$ `res`) found in the invariants, respectively.

In total, SLING generates 3214 invariants, consisting of 309 preconditions, 2442 postconditions and 463 loop invariants, in 487 target locations (average 6.60 invariants per target location). The total run time of SLING is 3866.06s for 149 programs (average 25.95s per program and 1.2s per invariant). The time to run the program and collecting traces is negligible (about a second for all programs).

Out of 157 programs, we were not able to obtain any traces for 5 programs (with *). These programs contain bugs that immediately result in runtime errors such as segmentation faults (thus we obtained no traces and inferred no invariants). For 15 programs (italic text), we were not able to obtain traces at all target locations and therefore inferred no invariants at those locations. These programs contain certain `return` branches that we could not reach using random inputs. For 3 programs (with †), we were not able to generate any loop invariants (though we were able to generate pre/post conditions for these programs). The SMT tool Z3 appears to stop responding when checking generated formula against traces collected at these locations. Finally, for 17 programs (bold text), we obtained *invalid* traces and therefore generated *spurious* invariants. This is an interesting behavior of running C programs and the LLDB debugger: a `free(x)` statement does not immediately free the pointer `x` so we can still observe (now invalid) heap values of `x` in the execution traces. Thus we conservatively consider all generated invariants depending on these traces spurious and report them in Table 1.

For other programs (and those where we only obtain traces at certain locations), we manually analyzed and compared SLING's generated invariants to documented ones. First, we found that *all* generated invariants are correct, i.e., they are true invariants at the considered locations. Thus, the

Table 1. Experimental results. Programs denoted with * contain bugs preventing us to obtain traces. Programs denoted with † cause SLING to timeout at certain locations. *Italic* programs have locations that cannot be reached using random inputs. **Bold** programs contain locations with `free` statements that give invalid traces. SLL and DLL stand for Singly and Doubly Linked Lists, respectively.

Programs	Total						Avg. Per Inv		
	LoC	iLocs	Traces	Invs	A/S/X	Time(s)	Single	Pred	Pure
SLL (8): append, delAll, find, insert, reverse, insertFront, insertBack, copy	168	26	226	30	8/0/0	40.54	0.37	0.83	1.03
Sorted List (10): concat, find, findLast, insert, insertIter, delAll, reverseSort, insertionSort, mergeSort, quickSort*	268	25	194	82	9/0/1	137.32	0.39	2.40	0.67
DLL (12): append, concat, meld, delAll, insertBack, insertFront, midInsert, midDel, midDelError, midDelHd, midDelStar, midDelMid	160	31	168	238	12/0/0	399.12	0.46	1.68	3.93
Circular List (4): insertFront, insertBack, ***delFront***, ***delBack***	97	11	14	42(16)	2/2/0	11.43	0.81	1.19	2.10
Binary Search Tree (5): *del*, findIter, *find*, insert, rmRoot*	144	16	66	24	2/2/1	24.02	0.50	1.21	1.54
AVL Tree (4): *avlBalance*, *del*, findSmallest, insert	194	13	56	37	2/2/0	22.12	1.22	0.57	3.08
Priority Tree (4): *del*, *find*, *insert*, *rmRoot*	154	19	64	273	2/2/0	341.37	3.30	1.66	3.30
Red-black Tree (2): del*, *insert*	287	11	70	63	0/1/1	44.8	2.10	1.08	8.11
Tree Traversal (5): traverseInorder, traversePostorder, traversePreorder, tree2list, tree2listIter*	168	12	174	12	4/0/1	22.93	0.08	0.58	0.50
glib/glist_DLL (10): find, ***free***, index, last, length, nth, nthData, position, prepend, reverse	216	31	128	435(20)	9/1/0	403.13	0	2.61	7.29
glib/glist_SLL (22): append, concat, copy, ***delLink***, find, ***free***, index, insertAtPos, *insertBefore*, *insertSorted*, last, length, nth, nthData, position, prepend, rm, rmAll, rmLink, reverse, sortMerge, *sortReal*	606	69	299	382(11)	17/5/0	879.35	0.56	2.28	2.07
OpenBSD Queue (6): init, insertAfter, insertHd, insertTl, ***rmAfter***, ***rmHd***	105	12	12	27(4)	4/2/0	10.04	0.15	2.04	0.15
Memory Region (1): memRegionDllOps	67	7	14	52	1/0/0	17.70	0.73	0.81	7.96
Binomial Heap (2): *findMin*, *merge*	117	8	54	89	0/2/0	76.56	1.39	0.90	9.15
SV-COMP (Heap Programs) (7): allocSlave, insertSlave, createSlave, destroySlave, add, del, init	119	16	34	71	7/0/0	58.17	0.24	1.66	3.41
GRASShopper_SLL (Iterative) (8): concat, copy, ***dispose***, filter, insert, ***rm***, reverse, traverse	193	27	111	98(9)	6/2/0	71.03	0.17	2.72	1.15
GRASShopper_SLL (Recursive) (8): concat, copy, ***dispose***, filter, insert, ***rm***, reverse, traverse	173	24	118	40(3)	6/2/0	30.94	0.28	2.00	1.1
GRASShopper_DLL (8): concat, copy, ***dispose***, filter†, insert, ***rm***, reverse, traverse	209	24	108	638(20)	5/2/1	803.58	0.04	2.95	8.50
GRASShopper_SortedList (14): concat, copy, ***dispose***, filter, insert, reverse, ***rm***, split, traverse, merge, doubleAll, pairwiseSum, insertionSort†, mergeSort*	394	43	195	222(1)	10/2/2	160.1	1.04	2.27	4.29
AFWP_SLL (11): create, ***delAll***, find, last, reverse, rotate, swap, insert, del†, *filter*, *merge*	264	25	89	94(11)	7/3/1	71.04	0.18	1.73	1.85
AFWP_DLL (2): dll_fix, dll_splice	40	5	16	133	2/0/0	75.51	0.02	2.96	6.67
Cyclist (4): *aplas-stack*, *composite4*, *iter*, schorr-waite	506	32	360	132	1/3/0	165.26	0.27	0.63	0.67

spurious results reported in Table 1 are only those caused by invalid traces as described above. Second, our results either matched (either syntactically or semantically equivalent) or, in many cases, were stronger than the documented invariants. For example, for `SLL/reverse`, we inferred the documented postcondition `sll(res)` and the additional constraints $x = \mathsf{nil} \wedge x = \mathsf{tmp}$ showing that the header of the input list x becomes the tail of the resulting list. In many similar cases, we achieved stronger results by inferring both the expected invariants and additional equalities.

A potential reason for these sound results is because we only infer shape properties using predicates and pure equalities. We also do not consider general disjunctive invariants or numerical relations (e.g., only check equivalences among memory addresses and do not consider other relationships such as the address of `x` is greater than that of `y`). Existing numerical invariant studies [37, 38] have shown that dynamic analysis often produces many spurious invariants involving disjunctions and general inequalities and thus focus on invariants that are conjunctions of equalities or inequalities under specific templates (e.g., octagonal invariants).

Although we tried our best to carefully check all generated results, the process of checking many complex SL invariants manually is time-consuming and difficult. In future work, we will use an automatic verifier that supports SL formulas to check SLING's invariants (see additional details in Section 6). Moreover, we might be able to leverage test-input generation techniques, e.g., symbolic execution with lazy-initialization [28] or SL predicates [27], to construct smart inputs, which can explore hard-to-reach program paths to infer better invariants.

5.4 Uses of Inferred Invariants

Dynamically inferred invariants can help users understand programs (e.g., generating loop invariants, pre/post conditions for unknown programs as shown in Section 2) and gain confidence about expected properties (e.g., the generated invariants met the expectation). Existing works also show many other uses of dynamic invariants including documentation, complexity analysis, fault localization, and bug repair [1, 15, 36, 44]. Below we list several concrete examples describing the usefulness of SLING's invariants.

Detecting Bugs. SLING's invariants can help reveal and explain bugs. For `Red-black Tree/insert`, we obtained an invariant that appears too "simple". Manual inspection showed that the inferred invariant is indeed correct: the program always crashes after the first iteration, thus the inferred invariant only captures a portion of data operated during the first iteration. For `glib/glist_SLL/sortMerge`, SLING reported an unexpected postcondition stating that the result is always null. Manual inspection revealed this is correct and is due to a (typo) bug in the program that returns `list_next` instead of `list->next`. For `AFWP/dll_fix.c`, the expected loop invariant is $\exists u_1, u_2, u_3, u_4.\ \texttt{sll}(i) * \mathsf{dll}(j, u_1, k, u_2) * \mathsf{dll}(k, u_3, u_4, \mathsf{nil})$, but SLING returned $\texttt{sll}(i) \land i = h \land k = j \land k = \mathsf{nil}$. Thus, the expected invariant shows that k can be non-nil, but SLING's invariant shows the opposite. Manual inspection showed a (potentially seeded) bug, where a guard checking for $k = \mathsf{nil}$ was commented out. Indeed, with this guard uncommented, SLING inferred the expected invariant.

Identifying Spurious Warnings. SLING's invariants can also help check results from static analyzers, e.g., to understand and gain confidence about reported results or detect potential problems. The Facebook Infer (FBInfer) [17] mentioned in Section 1 is a well-known SL static analyzer that produces warnings for memory safety bugs for iOS and Android apps. However, FBInfer can also produce spurious (false positive) warnings. For example, when analyzing the correct version of the mentioned `glib/glist_SLL/sortMerge` program, FBInfer reported a memory leak after the assignment `l->next =NULL;` at the end of a loop because it thinks that `l->next` is not reachable. However, SLING's inferred invariants at that location showed `l->next` is a valid alias to other pointer variables and reachable. Manual inspection confirmed that SLING's generated invariants are correct and the program has no memory leak at that location. We applied the same technique and found similar spurious warnings from FBInfer for 7 other programs[5].

Note that FBInfer also reported an error at another location of `sortMerge`. However, this time, SLING's invariants confirmed the warning and even revealed that the error is caused by a dangling pointer.

[5] `Binomial_Heap/merge`, `Circular_List/delBack`, `GRASShopper_DLL/insert`, `GRASShoper_SLL(Iter)/insert`, `GRASShoper_SortedList/insert`, and `copy`, `insert` in `Grasshopper_SLL (Rec)`.

Table 2. Comparing SLING to S2.

Programs	Total	Both	S2	SLING	Neither
SLL	9	8	0	1	0
Sorted List	14	6	0	6	2
DLL	13	0	0	13	0
Circular List	4	0	0	2	2
Binary Search Tree	6	1	1	2	2
AVL Tree	4	0	0	2	2
Priority Tree	4	1	1	1	1
Red-black Tree	2	0	0	0	2
Tree Traversal	6	3	0	2	1
glib/glist_DLL	19	0	0	18	1
glib/glist_SLL	40	6	0	29	5
OpenBSD Queue	6	0	0	4	2
Memory Region	3	1	0	2	0
Binomial Heap	2	0	1	0	1
SV-COMP	9	0	0	9	0
GRASShopper_SLL(Iter)	16	2	0	12	2
GRASShopper_SLL(Rec)	8	3	2	3	0
GRASShopper_DLL	16	0	0	13	3
GRASShopper_SortedList	29	1	0	24	4
AFWP_SLL	20	1	0	15	4
AFWP_DLL	3	0	0	3	0
Cyclist	4	0	0	1	3
Total Sum	237	33	5	162	37

5.5 Comparing to S2

We compare SLING to the static tool S2 [29], which uses the state-of-the-art *bi-abduction* technique [9] in SL to generate invariants proving memory safety of C programs, e.g., without null dereferencing and memory leaks. In addition to memory checking, S2 attempts to find strongest specifications consisting of pre/post conditions for heap programs.

We compare S2 to SLING using the same C benchmark programs[6] listed in Table 1. S2 only supports shape invariants, thus we only compare shape invariants generated by the two tools and ignore the pure invariants generated by SLING. Moreover, S2 does not infer invariants at arbitrary locations like SLING, instead it attempts to find complete specifications (involving both pre and postconditions) and loop invariants. Thus, we do not consider invariants generated at individual locations as shown in Table 1 and instead consider specifications as a whole and loop invariants. Note that each program has a specification but only programs with loops have loop invariants. As with SLING, we manual analyze the results of S2 and compare them to SLING.

Table 2 shows the results of S2 comparing to SLING. Column **Programs** lists the program categories, similarly to

[6] Several of these programs, e.g., those in SLL, DLL, and Binary trees, are also used in [29] to evaluate S2's capability of proving memory safety (but not its capability of finding strongest specifications).

those listed in Table 1. Column **Total** lists the number documented properties consisting of specifications and loop invariants for the programs in the corresponding category. The next four columns list the respective numbers of properties that **Both** tools can generate, **S2** can generate but SLING cannot, **SLING** can generate but S2 cannot, and **Neither** tools can generate. For example, the 10 programs in `Sorted List` have 14 properties (10 specifications and 4 loop invariants), from which there are 6 properties that both tool found, none that only S2 found, 6 that only SLING found, and 2 that both fail to find. Finally, S2 takes less than a second for all but the 4 `concat` programs in the `GRASShopper` categories, from which S2 appears to freeze.

The last row of Table 2 summarizes the results. First, both tools discovered and failed about 14% and 15% of the properties, respectively. Properties both tools found are from relatively simple recursive programs with singly-linked lists and trees. We do not observe any patterns among the programs containing properties that neither tool found, e.g., some failed results are caused by different reasons, e.g., for `SLL/quicksort`, S2 did produce any specification while SLING failed to generate any properties due the program crashed and produced no traces. Next, S2 found 5 properties that SLING did not. These properties mostly are in programs that we discussed in Section 5.3, in which SLING obtained incomplete or spurious results due to lack of traces, e.g., `binary_search_tree/find_rec.c`, or debugger "free" problem, e.g., `GRASShopper/rec/dispose.c`. Finally, SLING found many properties that S2 did not (162/237). For these properties, which often come from relatively complex programs with rich data structures, S2 either completely failed to produce them or produced much weaker than expected results.

In summary, SLING outperforms S2 by finding more than 65% of the total properties. We find this result encouraginging as it shows the competitiveness of dynamic analysis of SLING to static analysis.

6 Related Work

SLING is inspired by the well-known dynamic invariant generation tool Daikon [14, 15]. Daikon comes with a large list of invariant templates and predicates, tests them against program traces, removes those that fail, and reports the remaining ones as candidate invariants. Recently, several techniques (such as PIE [41], ICE [19], DIG [37], and SymInfer [36]) have been developed to infer numerical invariants using a hybrid approach that dynamically infers candidate invariants and then statically checks them against the program code. In general, these approaches do not consider SL invariants for memory shape analysis.

Program analysis in SL has rapidly gained adoption from both academia and industry in the past decade. The static analyzers MemCAD [22] and THOR [33, 34] can reason both shape and numerical properties of programs, but generate invariants for a restricted language of list manipulating programs [32]. Facebook Infer [8, 17] infers invariants to detect real memory bugs, but only supports simple data structures (e.g., linked lists) and restricted language features (e.g., no arithmetic). Using the similar bi-abduction technique as Infer, the static analysis tools CABER [6] and the S2 tool described in Section 5.5 offer more general, but also more expensive, approaches to infer shape properties. SLING infers invariants at arbitrary locations while CABER only generates preconditions. Moreover, SLING can generate invariants supported as well as those unsupported by these tools (as shown in Section 5.5).

The data-driven Locust tool [5] hybridizes dynamic and static analyses to infer invariants for programs written in a restricted and simple language. To infer an invariant, Locust expands the syntax of an SL formula using a machine learning model trained from a large set of data. Locust iteratively refines the inferred invariants using counterexamples obtained by the Grasshoper static verifier [45]. Locust is mainly evaluated on example programs with singly-linked lists and binary trees and does not support more complex data structures (e.g., the tool does not support doubly-linked list and returns no results when applied to our `concat` example). The tool also relies on an expensive training process over large data sets (e.g., Locust's training process took 43 minutes on our experiment machine described in Section 5).

Finally, automatic verification tools such as HIP [10], Grasshopper [45], Verifast [24], and VCDryad [43] can prove given SL specifications and invariants in heap-based programs. In future work, we intend to use these tools to automatically check SLING's inferred invariants.

7 Conclusion

We present SLING, a new dynamic analysis approach to inferring invariants in separation logic to reason about programs that manipulate data structures. SLING uses a debugger to obtain rich program execution traces, consisting of stack and heap contents. It then iterates over each program variables to analyze separating memory regions related that variable and model these regions using an invariant over predefined SL predicates. After obtaining individual invariants modeling separating memory regions, SLING combines them into a final SL formula describing the shape of explored memory regions. Preliminary results on a large set of nontrivial programs show that SLING is effective in discovering useful invariants describing operations over a wide variety of data structures. We believe that SLING takes an important step in broadening the space of properties about heap programs that can be dynamically inferred and exposes opportunities for researchers to exploit new dynamic SL invariant analyses.

References

[1] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *International Symposium on Software Testing and Analysis*. ACM, 177–188.

[2] Thomas Ball and Sriram K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Symposium on Principles of Programming Languages (Principles of Programming Languages)*. ACM, New York, NY, USA, 1–3. https://doi.org/10.1145/503272.503274

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2004. A Decidable Fragment of Separation Logic. In *Foundations of Software Technology and Theoretical Computer Science*. 97–109.

[4] Dirk Beyer. 2017. Software Verification with Validation of Results. In *TACAS*. Springer, 331–349.

[5] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *Static Analysis Symposium*. 66–87.

[6] James Brotherston and Nikos Gorogiannis. 2014. Cyclic Abduction of Inductively Defined Safety and Termination Preconditions. In *Static Analysis*. 68–84.

[7] James Brotherston, Nikos Gorogiannis, Max I. Kanovich, and Reuben Rowe. 2016. Model checking for symbolic-heap separation logic with inductive predicates. In *Symposium on Principles of Programming Languages*. 84–96.

[8] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*. 3–11.

[9] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66.

[10] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.* 77, 9 (2012), 1006–1036. https://doi.org/10.1016/j.scico.2010.07.004

[11] Byron Cook, Christoph Haase, Joël Ouaknine, Matthew J. Parkinson, and James Worrell. 2011. Tractable Reasoning in a Fragment of Separation Logic. In *CONCUR*. 235–249.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[13] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18 (1975), 453–457. Issue 8.

[14] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering* 27, 2 (2001), 99–123.

[15] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* (2007), 35–45.

[16] Daniel Fava, Julien Signoles, Matthieu Lemerre, Martin Schäf, and Ashish Tiwari. 2015. Gamifying program analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 591–605.

[17] fbinfer 2018. The Infer Static Analyzer. http://fbinfer.com/.

[18] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity*. 500–517.

[19] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Principles of Programming Languages*. ACM, 499–512.

[20] GDB 2018. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[21] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[22] Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2017. A Relational Shape Abstract Domain. In *NASA Formal Methods*. 212–229. https://doi.org/10.1007/978-3-319-57288-8_15

[23] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning about Reachability in Linked Data Structures. In *Computer Aided Verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 756–772.

[24] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*. 41–55. https://doi.org/10.1007/978-3-642-20398-5_4

[25] JDB 2018. jdb - The Java Debugger. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/jdb.html.

[26] Neil D. Jones and Steven S. Muchnick. 1982. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *Principles of Programming Languages*. ACM, 66–74.

[27] JSF 2018. JSF: The Java StarFinder Symbolic Execution Tool. https://github.com/star-finder/jpf-star.

[28] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 553–568.

[29] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. 2014. Shape Analysis via Second-Order Bi-Abduction. In *Computer Aided Verification*. 52–68.

[30] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages*. ACM, 42–54.

[31] LLDB 2018. The LLDB Debugger. https://lldb.llvm.org/.

[32] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring invariants in separation logic for imperative list-processing programs. *SPACE* 1, 1 (2006), 5–7.

[33] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2008. THOR: A Tool for Reasoning about Shape and Arithmetic. In *Computer Aided Verification*. 428–432.

[34] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2010. Automatic numeric abstractions for heap-manipulating programs. In *Symposium on Principles of Programming Languages*. 211–222. https://doi.org/10.1145/1706299.1706326

[35] Juan Antonio Navarro Pérez and Andrey Rybalchenko. 2011. Separation Logic + Superposition Calculus = Heap Theorem Prover. In *Programming Language Design and Implementation*. 556–566.

[36] ThanhVu Nguyen, Matthew Dwyer, and William Visser. 2017. SymInfer: Inferring Program Invariants using Symbolic States. In *Automated Software Engineering (ASE)*. IEEE, 804–814.

[37] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *Transactions on Software Engineering Methodology (TOSEM)* 23, 4 (2014), 30:1–30:30.

[38] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Using Dynamic Analysis to Generate Disjunctive Invariants. In *International Conference on Software Engineering (ICSE)*. IEEE, 608–619.

[39] Peter W. O'Hearn. 2016. CurryOn '16 Talk: Move fast to fix more things.

[40] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *International Conference on Computer Science Logic (CSL)*. 1–19.

[41] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven Precondition Inference with Learned Features. In *Programming Language Design and Implementation*. ACM, 42–56.

[42] PDB 2018. pdb - The Python Debugger. https://docs.python.org/2/library/pdb.html.

[43] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. 2014. Natural Proofs for Data Structure Manipulation in C Using Separation Logic. In *Programming Language Design and Implementation*. ACM, New York, NY, USA, 440–451. https://doi.org/10.1145/2594291.2594325

[44] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*. ACM, 87–102.

[45] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*. 124–139.

[46] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Symposium on Logic in Computer Science*. 55–74.

[47] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems* 24, 3 (2002), 217–298.

[48] VCDryad 2018. Automated deductive verification framework. http://madhu.cs.illinois.edu/vcdryad/.