Symbolic Execution for Memory Consumption Analysis

Duc-Hiep Chu Joxan Jaffar Rasool Maghareh

National University of Singapore, Republic of Singapore
{hiepcd, joxan, rasool}@comp.nus.edu.sg

Abstract

With the advances in both hardware and software of embedded systems in the past few years, dynamic memory allocation can now be safely used in embedded software. As a result, the need to develop methods to avoid heap overflow errors in safety-critical embedded systems has increased. Resource analysis of imperative programs with *non-regular* loop patterns and *signed* integers, to support both memory allocation and deallocation, has long been an open problem. Existing methods can generate symbolic bounds that are parametric w.r.t. the program inputs; such bounds, however, are imprecise in the presence of non-regular loop patterns.

In this paper, we present a worst-case memory consumption analysis, based upon the framework of symbolic execution. Our assumption is that loops (and recursions) of to-be-analyzed programs are indeed bounded. We then can exhaustively unroll loops and the memory consumption of each iteration can be precisely computed and summarized for aggregation. Because of path-sensitivity, our algorithm generates more precise bounds. Importantly, we demonstrate that by introducing a new concept of *reuse*, symbolic execution scales to a set of realistic benchmark programs.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; B.2.2 [*Performance Analysis and Design Aids*]: Verification,Worst-case analysis

Keywords Memory Consumption Analysis, Symbolic Execution, Interpolation, Reuse

1. Introduction

Traditionally, in safety-critical embedded systems, it was recommended not to use dynamic memory allocation because of two main reasons: (a) the allocation instructions might take longer than expected, resulting in the failure of temporal constraints in hard real-time systems; and (b) the memory fragmentation issue. As a result, stack was the only memory that grows dynamically during execution. Worst-case stack usage was estimated by methods such as the one proposed in [21]; the estimate is compared with the available memory to ensure stack overflow errors does not occur.

In the past few years, there have been advances in both hardware and software of embedded systems. The drop in the hardware cost, the development of customized operating systems for embedded systems, and finally the advent of constant time memory allocation algorithms with a reasonable handling of memory fragmentation [26, 27] are among these advances. Besides these, as the embedded systems become more complex, the need to use third-party code – which might require dynamic memory allocation – becomes more inevitable. As a result, dynamic memory allocation has now been used more frequently in embedded software [3].

Such increased use of dynamic memory allocation will raise concerns about the reliability of embedded systems that are deployed for safety-critical tasks. Thus, there is a real need for developing *program analysis* methods to avoid both stack and heap overflows in safety-critical systems. Besides, the estimate produced by such analyzers would be useful in the design process of embedded systems to reduce hardware cost [32]; it can also be presented to the programmers who are interested in dissecting the memory footprint of an embedded system.

Memory is a *non-cumulative resource*: what is acquired can later be released. As a result, unlike time and energy where the maximum consumption of an execution path is at the end of the path, the maximum memory usage of a path can be at any place in that path, e.g. right in the middle of it. Thus, many approaches developed for worst-case analysis of cumulative resources, such as WCET analysis, becomes inapplicable. More specifically, these methods often abstract away the orders between the acquires/releases, which is crucial for precise analysis of non-cumulative resource.

There has been a large body of work for *automatically* deriving symbolic upper bounds of memory consumption. Such analyses can provide a bound even when the program loops or recursions are not statically bounded. A bound generated by these methods is *parametric* in two types of program inputs: (1) the inputs that determine the maximum depths of the loops and recursions; and (2) the other program inputs. It is worth to note that the generated bound is often a *non-linear* formula over the first type of inputs.

Resource analysis of imperative programs with non-regular loop patterns and *signed* integers, to model both memory allocation and deallocation, has long been an open problem. By "non-regular", we mean that the loop does not behave *uniformly* across different iterations. We now mention some notable related work.

COSTA [1, 2], formulating the problem using the framework of cost relations, can infer parametric upper-bounds on the memory consumption of Java programs with region-based garbage collection. It was then extended [16] to generate more refined cost relations. On the other hand, [9] performs amortized resource analysis for C programs. However, these methods are quite limited in coping with non-linear formulas, in the sense that either the bounds generated are too imprecise or they have to require manual user interaction. They are further challenged by the programs of which the termination can only be decided if path-sensitivity is carefully taken into account.

In this paper, we present a worst-case memory consumption analysis, based upon the framework of *symbolic execution*. Our assumption is that loops (and recursions) of to-be-analyzed programs are indeed statically bounded. This is often the case in practice.

Our analysis is intra-procedural, but mainly for presentation purpose. As a summarization-based (thus compositional if needed) approach, it can be easily extended to inter-procedural, by also summarizing at the boundaries of functions, as opposed to only at the boundaries of loop iterations.

We adopt from [11], to symbolically and exhaustively unroll program loops. The memory consumption of each iteration can be *precisely* computed and summarized for aggregation. The key result from [11] is that non-regular loop patterns can be analyzed efficiently, often in a linear number of steps.

A bound generated by our analysis still is symbolic, but this is mainly because programs can allocate and/or deallocate a non-fixed amount of memory, e.g. via some input variable that is not statically determined. Our bound, however, is not parametric w.r.t. program inputs dictating the maximum depths of the program loops. As a result, we do not need to deal with the challenging problem of inferring closed-form expressions for the loops. This enables our method to have a higher level of automation, while producing more accurate bounds.

In detail, given a program, our analysis starts by constructing the symbolic execution tree, from which an estimate of the worst-case memory consumption can be easily extracted. Being highly path-sensitive, our analysis disregards infeasible combinations of allocations/deallocations from consideration, thus producing accurate bounds. In [13], Chu et al. have introduced the concept *reuse with interpolation and dominance* to achieve scalable symbolic execution for integrated timing analysis. The presence of *cache* makes the worst-case timing of each basic block *dynamic*, i.e. dependent on the contexts where the block is executed. In this paper, though we adopt their method for scalability, we still need to address two major challenges:

- First, it is the issue of *non-cumulative* resource. This requires the interplay between the *net* usage and the *high-water mark* usage of memory. As will be shown later, to accommodate this, we need to introduce a new component in our summarization.

- Second, it is the issue of dealing with symbolic cost of an instruction, as opposed to concrete cost in many related work, as well as in [13]. The need to compare between symbolic expressions leads us to the usage of the standard MAX function. Importantly, how it is used in tandem with the capturing of "dominance conditions" is one key contribution of this paper. We elaborate more in Section 3.

2. Overview Examples

Consider the C program from Figure 1. The program points are shown in brackets, e.g. ⟨1⟩. The allocations (deallocations) in the program are annotated with increment (decrement) statements (in red color). The resource variable r captures the resource of interest: memory. Note that this variable is always initialized to 0 and both increment and decrement operations are allowed on it.

Our analysis performs a depth-first traversal of the symbolic execution tree (of the program) and returns the highest value of r, possibly symbolic, that we can have along all the paths in the tree. Recall that because memory is a non-cumulative resource, the highest value of r can be at any place along a path (and in general *not* at the end of a path).

For the example in Figure 1, the highest value of r in the `then` branch is 10 (at ⟨5⟩) and in the `else` branch is 15 (at ⟨10⟩). These two values are compared at the parent node, namely ⟨3⟩. Because the highest value of r in the `else` branch is larger, we say the `else` branch *dominates* the `then` branch, under the current context. Then the path ⟨3⟩, ⟨8⟩, . . . , ⟨11⟩ is returned as the dominating path in the program. As a result, 15 – the highest value of r in the dominating path – is returned as a sound estimate for the worst-case memory consumption of the program.

```
⟨1⟩    int main(int j){
⟨2⟩        int n; r = 0;
⟨3⟩        if(j > 0){
⟨4⟩            n = 10;
⟨5⟩            char *matrix = malloc(n); r = r + n;
⟨6⟩            /* normal computation */
⟨7⟩            free(matrix); r = r - n;
           } else {
⟨8⟩            n = 5
⟨9⟩            char *matrix = malloc(n + 10); r = r + (n + 10);
⟨10⟩           /* normal computation */
⟨11⟩           free(matrix); r = r - (n + 10);
           }
⟨12⟩       return 0;
       }
```

Figure 1: An Annotated C program

Before stepping into a full analysis example, let us revisit the scalability issue. As stated in Section 1, one contribution of this paper is to adapt the concept of "reuse with interpolation and dominance" to the setting where the increment/decrement amount of the resource usage (memory) can be a *symbolic* value.

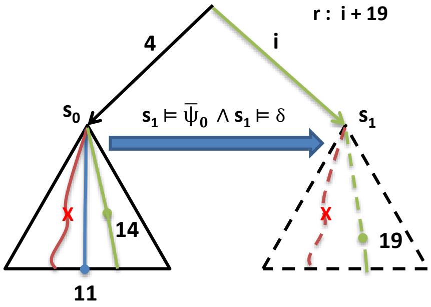

Figure 2: Reuse with Interpolation and Dominance

Figure 2 depicts a symbolic execution tree, where each triangle represents a subtree. The symbolic states s_0 and s_1 act as the program contexts for the left and right subtrees, respectively; and they are different visits to the *same program point*. Our analysis starts with exploring the left subtree. After traversing the left subtree, we obtain a summarization, comprised of four main components:

1) An *interpolant* Ψ_0, which is a generalization of s_0 that captures a condition preserving the infeasible paths of the subtree. Infeasible paths are marked with a red cross.

2) A "dominating" path, also called a *witness* path, denoted by Γ_h, which gives rise to the worst-case memory consumption of the subtree. This path is indicated in green color.

3) A second witness path, denoted by Γ_n, which captures the highest *net* memory usage at the end of the subtree. This path is indicated in blue color. The use of two witness paths is critical for safely combining summarizations, presented in Section 3.

4) A dominating condition δ, a formula which sufficiently guarantees that the dominating path *remains optimal*, i.e., the worst-case path in the subtree, when encountering a new context.

Considering we are now at s_1. Suppose that all the paths that were infeasible in s_0 stay infeasible, i.e. $s_1 \models \Psi_0$, and the dominating condition applies, i.e. $s_1 \models \delta$. This allows us to "reuse" the previous summarization. We then need to *replay* the witness path Γ_h under the context s_1. This, importantly, can lead to new *value* of the path (now 19), which is different from the original value (14). This is because the valuations of some symbolic expressions (or variables) are different, under the new context s_1, as opposed to the old context s_0 and may also hit the spike in another point along the path (as shown in the figure).

```
⟨0⟩ void main(int c){
⟨1⟩     assume(c ≥ 0); int N = 3;
        r = 0;
⟨2⟩     int** matrix = malloc(5 * sizeof(int*));
        r = r + 5 * 8;
⟨3⟩     char * b = (char*) malloc(c);
        r = r + c;
⟨4⟩     for (int i = 0; i ≤ N; i++){
⟨5⟩         if ((i % 2) == 0){
⟨6⟩             matrix[i]=malloc(i*sizeof(int));
                r = r + (i * 4);
            }
        }
⟨7⟩     free (b); r = r - c;
    }
⟨8⟩
```

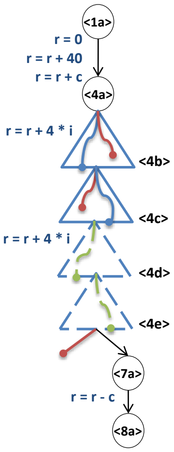

Figure 3: A Complicated Allocation Pattern (a); Our Analysis (b)

Backtracking to the root, and assuming that $i > 0$, thus $i + 19 > 4 + 14$. Therefore the right path in green is chosen as the overall dominating path. We then can conclude the analysis on the whole tree with the *symbolic* value $i + 19$.

Next, consider the C program in Figure 3(a) where we will discuss the concepts under the presence of a loop. Figure 3(b) depicts a symbolic execution tree of the corresponding program, where each triangle represents a loop iteration. For each node, in additional to a program point, we also use a letter to distinguish multiple visits to the same program point. An infeasible node is identified with a red bullet. For instance, at ⟨4e⟩, the red bullet indicates that it is not possible to re-enter the loop body. For readability, the program points ⟨2⟩ and ⟨3⟩ are not shown in the tree.

We note that loops are exhaustively unrolled and contexts (of feasible paths) are merged in the end of each loop iteration.

Starting the analysis, the value of r is successively increased by 40 and then by the value of c (input argument) from program points ⟨1⟩ to ⟨4⟩. In the first loop iteration, the `then` branch is feasible and the value of r is increased by the value of $i * 4$, which is 0. Note that the `else` branch (colored in red in the symbolic execution tree) is *infeasible* because $i = 0 \wedge i \;\%\; 2 \;\neq\; 0$ is equivalent to $false$.

At ⟨4b⟩, the analysis moves to the second iteration where the `then` branch is *infeasible*, because $(i = 1 \wedge i \;\%\; 2 \;\;== \;\;0)$ is equivalent to $false$. There are no allocation/deallocations in the `else` branch, thus the value of r is unchanged. Similarly, in the third and fourth iterations the value of r is increased by 8 ($2{*}4$) and 0, respectively. Finally, only at ⟨4e⟩, exiting the loop is possible, while re-entering is not. We continue with the node ⟨7a⟩, where r is then decreased by c. Because c is non-negative, the maximum value of r is reached at ⟨7a⟩ which is $48 + c$.

Now let us go a bit deeper into the technicality. After the traversal of the first iteration, the maximum increase/decrease in the value of r in the iteration, which is $+(i{*}4)$, was stored in a summarization of the loop iteration as a witness Γ_h. (The second witness path Γ_n coincides with Γ_h in this example.) Since there is only one feasible path in the loop iteration, the dominating condition is *true* and the interpolant stored in the summarization is $i \;\%\; 2 \;\neq\; 0$ which is enough to capture the reason of infeasibility.

In a following loop iteration where the interpolant and the dominating condition hold, for example at ⟨4c⟩, the summarization of the first iteration is then reused. The analysis of the new iteration can be deduced to be $+(8)$, without the need of exploring all other paths in the loop iteration.

We also note that this summarization cannot be applied to the second iteration at ⟨4b⟩. This is because the interpolant test fails. Fast forwarding, we finally mention that in Figure 3(b), the respective triangles of the third and the fourth iterations are shown in dotted lines to indicate that they were not explored in full. Instead, we reuse the summarizations of other iterations.

3. General Framework

We model a program by a transition system. A transition system $\mathcal{P}$ is a tuple $\langle \mathcal{L}, \ell_0, \longrightarrow \rangle$ where $\mathcal{L}$ is the set of program points, $\ell_0 \in \mathcal{L}$ is the unique initial program point. Let $\longrightarrow \subseteq \mathcal{L} \times \mathcal{L} \times Ops$, where Ops is the set of operations, be the transition relation that relates a state to its (possible) successors by executing the operations.

Basic operations are either assignments, "assume" operations or memory allocations/deallocations. The set of all program variables is denoted by $Vars$ including a special variable r tracking the amount of memory consumption. An assignment $x := e$ corresponds to assign the evaluation of the expression e to the variable x. The expression $assume(cond)$ means: if the conditional expression $cond$ evaluates to true, execution continues; otherwise it halts. Moreover, $alc(+, e)$ or $alc(-, e)$ corresponds to a memory allocation or deallocation, respectively, of size e. These operations are compiled from the `malloc` and `free` statements in the input C programs. We shall use $\ell \xrightarrow{op} \ell'$ to denote a transition relation from $\ell \in \mathcal{L}$ to $\ell' \in \mathcal{L}$ executing the operation $op \in Ops$. Clearly a transition system is derivable from a control flow graph (CFG).

Definition 1 (Symbolic State)**.** *A symbolic state s is a tuple $\langle \ell, \sigma, \Pi \rangle$ where $\ell \in \mathcal{L}$ is the current program point, the symbolic store σ is a function from program variables to terms over input symbolic variables, and finally the path condition Π is a first-order formula over the symbolic inputs.* □

Let $s_0 \overset{\text{def}}{=} \langle \ell_0, \sigma_0, \Pi_0 \rangle$ denote a unique initial symbolic state. At s_0, r is initialized to 0 while other program variables are initialized to fresh symbolic variables. For every state $s \equiv \langle \ell, \sigma, \Pi \rangle$, the evaluation $[\![e]\!]_\sigma$ of an arithmetic expression e in a store σ is defined as usual: $[\![v]\!]_\sigma = \sigma(v)$, $[\![n]\!]_\sigma = n$, $[\![e + e']\!]_\sigma = [\![e]\!]_\sigma + [\![e']\!]_\sigma$, $[\![e - e']\!]_\sigma = [\![e]\!]_\sigma - [\![e']\!]_\sigma$, etc. The evaluation of the conditional expression $[\![cond]\!]_\sigma$ can be defined analogously. The set of first-order logic formulas and symbolic states are denoted by *FO* and *SymStates*, respectively.

Definition 2 (Transition Step)**.** *Given $\langle \mathcal{L}, l_0, \longrightarrow \rangle$, a transition system, and a symbolic state $s \equiv \langle \ell, \sigma, \Pi \rangle \in SymStates$, the symbolic execution of transition $tr : \ell \xrightarrow{op} \ell'$ returns another symbolic state s' defined as:*

$$s' \overset{\text{def}}{=} \begin{cases} \langle \ell', \sigma, \Pi \wedge cond \rangle & \textit{if } op \equiv \textit{assume(cond)} \\ \langle \ell', \sigma[x \mapsto [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv x := e \\ \langle \ell', \sigma[r \mapsto r + [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv \textit{alc(+,e)} \\ \langle \ell', \sigma[r \mapsto r - [\![e]\!]_\sigma], \Pi \rangle & \textit{if } op \equiv \textit{alc(-,e)} \end{cases} \qquad \square$$

Abusing notation, the execution step from s to s' is denoted as $s \xrightarrow{tr} s'$ where tr is a transition. Given a symbolic state $s \equiv \langle \ell, \sigma, \Pi \rangle$ we also define $[\![s]\!] : SymStates \rightarrow FO$ as the projection of the formula

$$[\![\Pi]\!]_\sigma \wedge \bigwedge_{v \in Vars} v = [\![v]\!]_\sigma$$

onto the set of variables *Vars*. The projection is performed by the elimination of existentially quantified variables.

For convenience, when there is no ambiguity, we just refer to the symbolic state s using the abbreviated tuple $\langle \ell, [\![s]\!] \rangle$ where ℓ is as before, and $[\![s]\!]$ is obtained by projecting s as described above. A path $\pi \equiv s_0 \rightarrow s_1 \rightarrow \ldots s_m$ is feasible if $s_m \equiv \langle \ell, [\![s_m]\!] \rangle$ and $[\![s_m]\!]$ is satisfiable. Otherwise, the path is called *infeasible* and s_m is called an infeasible state. Here we query a *theorem prover* for satisfiability checking on the path condition. We assume the theorem prover is sound, but not complete. If $\ell \in \mathcal{L}$ and there is no transition from ℓ to another program point, then ℓ is called the *end point* of the program. Under that circumstance, if s_m is feasible, then s_m is called *terminal* state.

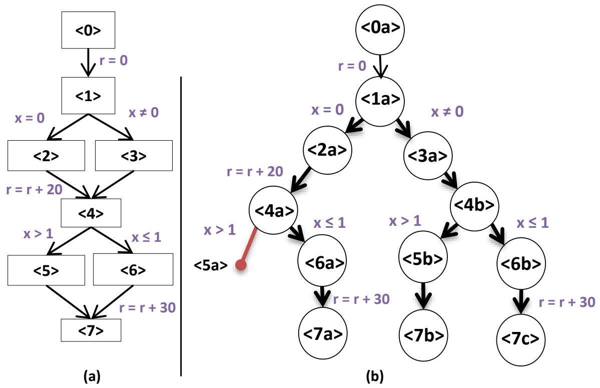

Figure 4: A CFG (a); and Its Symbolic Execution Tree (b)

Example 1 (Symbolic Execution Tree)**.** *Consider the CFG in Figure 4(a). Each node abstracts a basic block. In each basic block, a* program point *is shown. For brevity, we might use interchangeably the identifying program point when referring to a basic block. Two outgoing edges signify a branching structure, while the branch conditions are labeled beside the edges. Moreover, r is set to 0 in the beginning and the updates to it are also shown.*

Next, in Figure 4(b), we show our analysis tree*. Each node, shown as a circle, is identified by the corresponding program point, followed by a letter to distinguish between multiple visits to the same program point. Each path denotes a* symbolic execution path *of the program. Each node is associated with a symbolic state, but for simplicity we do not explicitly show any state content.*

Now assume that no basic block modifies x. At node $\langle 5a \rangle$, the projection of the path condition over program variables Vars*, namely $[\![s_{5a}]\!]$, is $r = 20 \wedge x = 0 \wedge x > 1$, which is equivalent to* false*. In other words, the leftmost path in Figure 4(b) is in fact* infeasible*. On the other hand, at node $\langle 7a \rangle$, the projection of the path condition over program variables* Vars*, namely $[\![s_{7a}]\!]$, is $r = 50 \wedge x = 0 \wedge x \leq 1$. As it can be seen, the high-water mark usage of memory is at $\langle 7a \rangle$ with the value of* 50*.*

Recall that our transition system is a directed graph. We now introduce relevant concepts for our loop unrolling framework. We assume that each loop has only one loop head and one unique end point. For each loop, following the back edge from the end point to the loop head, we do not execute any operation. These assumptions can be achieved by a preprocessing phase.

Definition 3 (Loop)**.** *Given a directed graph $G = (V, E)$ (our transition system), we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a loop of G.*

Definition 4 (Loop Head)**.** *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G, we call $\mathcal{E} \in V_L$ a loop head of L, also denoted by $\mathcal{E}(L)$, if no node in V_L, other than $\mathcal{E}$ has a direct successor outside L.*

Definition 5 (End Point of Loop Body)**.** *Given a directed graph $G = (V, E)$, a loop $L = (V_L, E_L)$ of G and its loop head $\mathcal{E}$. We say that a node $u \in V_L$ is an end point of a loop body if there exists an edge $(u, \mathcal{E}) \in E_L$.*

Definition 6 (Same Nesting Level)**.** *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$, we say two nodes u and v are in the same nesting level if for each loop $L = (V_L, E_L)$ of G, $u \in V_L \Longleftrightarrow v \in V_L$.*

A "subtree" is a portion of a symbolic execution tree. Given a state s and program point ℓ_2 such that (a) state $s \equiv \langle \ell_1, [\![s]\!] \rangle$ appears in the tree, and (b) ℓ_2 post-dominates ℓ_1, then $subtree(s, \ell_2)$ depicts all the paths emanating from s and, if feasible, terminate at ℓ_2. (Note that ℓ_2 may not be the end point of the whole tree.) We call ℓ_1 and ℓ_2 the entry and exit points of the subtree.

A summarization of a subtree, intuitively, is a *succinct description* of its analysis. This is formalized as a tuple of important components of the analysis. These are: the entry and exit program points, an interpolant describing infeasible paths, two witnesses, one describing the sub-path with the high-water mark of memory usage and one describing the sub-path with the highest net memory usage, two domination conditions ensuring each witness is, respectively, the worst-case sub-path in the subtree, and finally an abstract transformer relating the input and output variables.

We start with our notion of interpolant. The idea here is to approximate at the root of a subtree, the weakest precondition in order to maintain the infeasibility of all the nodes inside. (An exact computation is in general undecidable.) In the context of program verification, an interpolant captures succinctly a condition which ensures the *safety* of the tree at hand. Adapting this to program analysis is first done in [20]. Since all infeasible nodes are excluded from calculating the analysis result of a subtree, in order to ensure soundness, at the point of reusing the result, all such infeasibility must also be maintained.

Next, we discuss the concept of *witness*. A high-water mark witness is a sub-path from the root of the subtree $subtree(s, \ell_2)$ to a program point ℓ inside the subtree where the resource variable r reaches its peak value. It is depicted by $\Gamma_h \equiv \langle \gamma_h, \pi_h \rangle$ where γ_h is the sequence of $alc(\pm, e)$ depicting all memory allocation or deallocations and π_h is the path constraints along the witness.

The witness of highest net-usage is a sub-path from the root of the subtree $subtree(s, \ell_2)$ to the program point ℓ_2 where the resource variable r has the highest value at ℓ_2 and it is depicted by $\Gamma_n \equiv \langle \gamma_n, \pi_n \rangle$.

Note that the two witnesses can be different. Also, to reduce the size of the witness, consecutive allocations of concrete amount are merged into one. Similarly for consecutive deallocations.

Because non-constant allocations may be evaluated to different values, with different contexts. For such evaluation, we rely on the *estimating upper-bound* and the *estimating lower-bound* functions.

Definition 7 (Estimating Upper-bound (EUB))**.** *Given a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$ and a non-constant allocation of the amount captured by an expression e, the function* EUB$(e, [\![s]\!])$ *returns the smallest expression ub over symbolic input parameters and concrete values such that $[\![s]\!] \models e \leq ub$. In case no such upper-bound can be generated, e is returned.* □

Definition 8 (Estimating Lower-bound (ELB))**.** *Given a symbolic state $s \equiv \langle \ell, [\![s]\!] \rangle$ and a non-constant deallocation of the amount captured by an expression e, the function* ELB$(e, [\![s]\!])$ *returns the largest expression lb over symbolic input parameters and concrete values such that such that $[\![s]\!] \models lb \leq e$.* □

In summary, EUB and ELB are to over-estimate and under-estimate non-constant allocations and deallocations, respectively. Note that, in the worst case, ELB can always return 0 as a trivial lower bound. Generating sound, but not precise bounds using EUB or ELB would affect the overall precision of the analysis.

Example 2 (Witness Path)**.** *Assume the following sequence of allocating/deallocating statements along a symbolic path, which is selected as a high-water mark witness:*

$$x = \mathtt{malloc}(10), y = \mathtt{malloc}(5), \mathtt{free}(y), \mathtt{free}(x), z = \mathtt{malloc}(c)$$

This sequence would be stored as $\gamma_h \equiv ([+, 15], [-, 15], [+, c])$.

When the summarization is reused, the high-water mark memory usage is computed by replaying the sequence γ_h under the new incoming context s, making use of the EUB and ELB functions. For example, given that $[\![s]\!] \equiv c < 5$, γ_h in the above example will be approximated by $\gamma_1 = [+, 15], [-, 15], [+, 4]$, which gives us a high-water mark of 15. In a different context where $[\![s]\!] \equiv c < 20$, γ_h will be approximated by $\gamma_1 = [+, 15], [-, 15], [+, 19]$, which gives us a high-water mark of 19.

Finally, as in [11], the feasibility of a witness $\Gamma \equiv \langle \gamma, \pi \rangle$ w.r.t. to an incoming context s is determined by checking if $[\![\pi]\!] \wedge [\![s]\!]$ is satisfiable. In what follows, we abbreviate $[\![\pi]\!] \wedge [\![s]\!]$ by $[\![\Gamma]\!]$.

We say that two nodes in a symbolic execution tree are *similar* if they refer to the same program point. Thus, two subtrees are similar if they share the same entry and exit program points.

We next discuss dominating condition, another component of our analysis of a subtree. Each dominating condition is generated with respect to a witness. Intuitively, this answers the question "in what context of a similar subtree does the witness *remain optimal*?"

More specifically, the constraints in the dominating condition are typically of the form $x \leq y$ where x, y are either program variables or some concrete values — note that at least one must be a variable. The domination condition is computed by abstracting the context that gives rise to dominance in the first place.

We now discuss an abstract transformer Δ of a subtree from ℓ_1 to ℓ_2 which is an abstraction of all feasible paths (w.r.t. the incoming symbolic state s) from ℓ_1 to ℓ_2. Its purpose is to capture an input-output relation between the variables. In our implementation, we adopt from [11] which uses the polyhedral domain [14].

We collect the components discussed above into a definition.

Definition 9. *A* summarization *of* $subtree(s, \ell_2)$*, where* ℓ_1 *is the program point of* s*, is a tuple*

$$[\ell_1, \ell_2, \Psi, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta]$$

where Ψ *is an interpolant,* Γ_h *and* Γ_n *are the high-water mark and net-usage witnesses,* mhw *is the high-water mark memory usage of the subtree, and* δ_h *and* δ_n *are the respective dominating conditions. Finally,* Δ *is an abstract transformer relating the input and output variables.* □

We now display a key feature of our algorithm: reuse of a summarization. Suppose we have already computed a summarization $[\ell_1, \ell_2, \Psi, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta]$ where the high-water mark witness is $\Gamma_h \equiv \langle \gamma_h, \pi_h \rangle$. Suppose we then encounter a symbolic state $s' \equiv \langle \ell_1, [\![s']\!] \rangle$. The summarization can be reused if:

1. $[\![s']\!]$ implies the stored interpolant Ψ i.e., $[\![s']\!] \models \Psi$.
2. The context of s' is consistent with the witness formula, i.e., $[\![\pi_h]\!] \wedge [\![s']\!]$ is satisfiable.
3. The high-water mark dominating condition is satisfied by s', i.e., $[\![s']\!] \models \delta_h$ holds.

The worst-case heap memory consumption of the subtree beneath the state s' is the peak value of r derived from the witness γ_h w.r.t. the context $[\![s']\!]$. Note that this worst-case can be different from the mhw stored in the summarization.

We now conclude this section by mentioning that we only summarize at selected program points. Given entry point ℓ_1, the corresponding exit point ℓ_2 is determined as follows. It is the program point that post-dominates ℓ_1 s.t. ℓ_2 is of the same nesting level as ℓ_1 and either is (1) an end point of the program, or (2) an end point of some loop body. In other words, we only perform "merging" abstraction at loop boundaries. As ℓ_2 can always be deduced from ℓ_1, in a summarization, we omit the component about ℓ_2.

4. An Example Analysis

Figure 5(a) presents the CFG of an example program. Its symbolic execution tree is depicted in Figure 5(b). Both the CFG and the tree are annotated with the updates on the resource variable r (in red color). Assume that b is a symbolic input parameter for this example program. The variable of interest r is initialized to 0 between nodes $\langle 1a \rangle$ and $\langle 2a \rangle$. The value of r can be seen beside node $\langle 2a \rangle$ (in green color). Note that in Figure 5(b), we do not (fully) show the subtree below node $\langle 5b \rangle$ and that we do not discuss the abstract transformers in detail.

In the left-most path, the value of r is updated to 20 at $\langle 5a \rangle$, 0 at $\langle 8a \rangle$ and c at $\langle 11a \rangle$, which is a symbolic value. Note that c is a random number in the range of $[0, b-1]$ and cannot be determined statically. In the second path, reaching $\langle 10a \rangle$, the path constraints contains both $j = 0$ and $j < 0$ (only relevant constraints are shown for brevity), thus an infeasible path is detected.

After finishing the subtree beneath $\langle 8a \rangle$, the following summarization is computed and stored:

$$[\langle 8 \rangle, \Psi, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta],$$

where the stored interpolant is $\Psi \equiv j \geq 0$, which is a succinct reason for the infeasibility of the right sub-path; the stored dominating conditions δ_h and δ_n are $true$, given that there is only one feasible path. Similarly, the only feasible path (in blue color) is stored both as the high-water mark witness $\Gamma_h \equiv \langle ([+, c]), j \geq 0 \rangle$ and as the net-usage witness $\Gamma_n \equiv \langle ([+, c]), j \geq 0 \rangle$, where $j \geq 0$ is the witness path constraint and $([+, c])$ is the sequence of the allocation/deallocations along the witness path. Finally, mhw, the worst-case heap memory consumption of the subtree, is computed to $b - 1$, by evaluating the memory consumption of the high-water mark witness. This, in turn, is achieved by invoking the function EUB with c as the first argument and the current context as the second argument.

Continuing the analysis, consider the pair of nodes $\langle 8a \rangle$ and $\langle 8b \rangle$. At node $\langle 8b \rangle$, the value of r is 40, due to the memory allocation from $\langle 7a \rangle$ to $\langle 8b \rangle$. We will check the reuse conditions here. We *first* check whether the stored *interpolant* (Ψ) is implied. This does not hold. The key reason is: some infeasible path in $\langle 8a \rangle$ is in fact feasible in $\langle 8b \rangle$. As a result, reuse does not happen and the node $\langle 8b \rangle$ is expanded.

After the analysis of the subtree beneath $\langle 8b \rangle$, a summarization is generated from the analysis of node $\langle 8b \rangle$. The summarization would be $[\langle 8 \rangle, \Psi', \Gamma'_h, \Gamma'_n, mhw', \delta'_h, \delta'_n, \Delta']$. The stored interpolant Ψ' is simply $true$ because both paths in the subtree are feasible. Comparing the peak value of r along these two feasible sub-paths, it can be seen that the value of r is larger in the right sub-path. So, the right sub-path is chosen as the high-water mark witness and Γ'_h is stored as $\langle ([+, b], [-, b]), j < 0 \rangle$. Consequently, mhw' is set to b.

Now, we need to capture a dominating condition such that when it holds, it is guaranteed that the chosen witness path dominates all the other path(s) in the subtree. For any symbolic state s, it is the case that $[\![s]\!] \models \text{EUB}(c, [\![s]\!]) < \text{EUB}(b, [\![s]\!])$. Thus the stored dominating condition δ'_h is simply $true$.

On the other hand, the value of r at $\langle 11c \rangle$ is $40 + c$, which is higher than the value of r at $\langle 11d \rangle$, which is 40. So the net-usage witness is the sub-path $\langle 8b \rangle$, $\langle 9b \rangle$, $\langle 11c \rangle$ (with green color) $\Gamma'_n \equiv \langle ([+, c]), j \geq 0 \rangle$. Moreover, the net-usage dominating condition δ'_n would also be simply $true$. For brevity, we would not discuss net-usage witnesses and their dominating conditions in the rest of this example.

Continuing the analysis, consider the pair of nodes $\langle 5a \rangle$ and $\langle 5b \rangle$. We will show that reuse can in fact take place here. Please take note, without proof, that: (1) the high-water mark witness of the subtree rooted at $\langle 5a \rangle$ is the rightmost feasible path $\langle 5a \rangle$, $\langle 7a \rangle$,

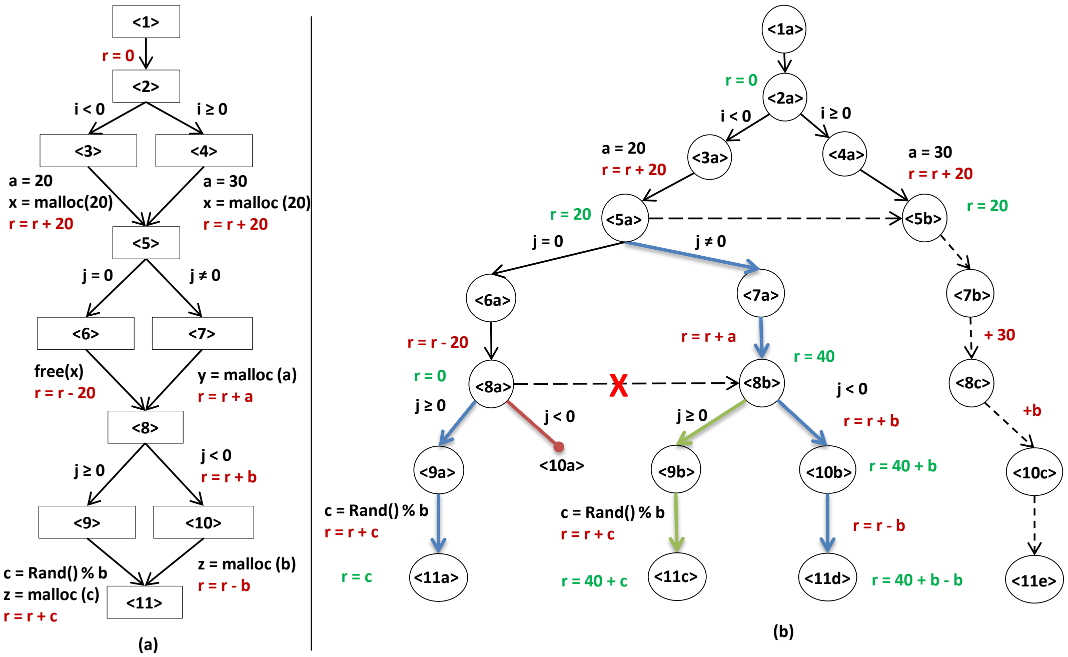

Figure 5: (a) The CFG of an annotated program (b) The analysis tree of the program

⟨8b⟩, ⟨10b⟩, ⟨11d⟩ (in blue color), stored as $\langle(([+,a],[+,b]), j < 0\rangle$; (2) The interpolant of interest is $true$; and the dominating condition for the high-water mark witness is also $true$.

Now we can exemplify the reuse at ⟨5b⟩. We first check if the context of ⟨5b⟩, called $[\![s_{5b}]\!]$ implies the interpolant computed after finishing ⟨5a⟩. In this case, the interpolant is $true$, thus the implication trivially holds. We then check whether the dominating condition, which is $true$, holds. This is also trivially satisfied, thus we can reuse the *high-water mark witness* of ⟨5a⟩, yielding an overall worst-case memory consumption of $20 + \text{EUB}(a, [\![s_{5b}]\!]) + \text{EUB}(b, [\![s_{5b}]\!])$, which is simplified to be $50 + b$.

We remark here that, the worst-case consumption of the subpath ⟨5⟩ ⟨7⟩ ⟨8⟩ ⟨10⟩ ⟨11⟩ for contexts ⟨5a⟩ and ⟨5b⟩ are indeed different. It is because, fundamentally, the worst-case consumption of a symbolic path is dependent on its context. In this particular example, the valuation of a in the two contexts is different.

5. Symbolic Execution with Reuse

For brevity, in the pseudocode we use F and T as abbreviations of $false$ and $true$, respectively. We use $\cdot$ to represent a component that is *not* of interest.

Algorithm 1 consists of two important functions. The function Analyze takes as input the initial symbolic state s_0 and the transition system $\mathcal{P}$ of an input program. It then invokes Summarize to generate a summarization for the whole program (line 1) and returns mhw, stored in the summarization, as the high watermark memory usage of the whole program (line 2).

The function Summarize performs a depth-first traversal of the symbolic execution tree. During the traversal, at each node either (1) a summarization is reused, thus we do not need to expand the node; or (2) after expanding it, we compute its summarization based on the summarizations of its child nodes. We now discuss the Summarize function in detail.

Base Cases: Summarize handles 4 base cases. First, when the symbolic state s is infeasible (line 3), no execution needs to be considered. Note that here path-sensitivity plays a role since only provably executable paths will be considered. As a result, the returned

Algorithm 1 The Analysis Algorithm

```
function Analyze($s_0$, $\mathcal{P}$)
⟨1⟩ $[\ell_0, \cdot, \cdot, \cdot, mhw, \cdot, \cdot, \cdot]$ := Summarize($s_0, \mathcal{P}$)
⟨2⟩ return $mhw$

function Summarize($s$, $\mathcal{P}$)
    Let $s$ be $\langle \ell, [\![s]\!] \rangle$
⟨3⟩ if ($[\![s]\!] \equiv F$)
        return $[\ell, F, \langle(([-,\infty]), F\rangle, \langle(([-,\infty]), F\rangle, -\infty, F, F, F]$
⟨4⟩ if (outgoing($\ell, \mathcal{P}$) $= \emptyset$)
        return $[\ell, T, \langle(([+,0]), T\rangle, \langle(([+,0]), T\rangle, 0, T, T, Id]$
⟨5⟩ if (loop-end($\ell, \mathcal{P}$)) return
        return $[\ell, T, \langle(([+,0]), T\rangle, \langle(([+,0]), T\rangle, 0, T, T, Id]$
⟨6⟩ $S := [\ell, \Psi, \Gamma_h, \Gamma_n, w, \delta_h, \delta_n, \Delta]$ := memoed($\ell$)
⟨7⟩ if ($S \neq \emptyset \wedge [\![s]\!] \models \Psi \wedge [\![\Gamma_h]\!] \not\equiv F \wedge [\![s]\!] \models \delta_h$) return $S$
⟨8⟩ if (loop-head($\ell, \mathcal{P}$))
⟨9⟩     $S_1 := [\cdot, \cdot, \Gamma_{h1}, \cdot, \cdot, \cdot, \cdot, \Delta]$ := TransStep($s, \mathcal{P}$, entry($\ell, \mathcal{P}$))
⟨10⟩    if ($[\![\Gamma_{h1}]\!] \equiv F$) $\overline{S}$ := Join($s, S_1$, TransStep($s, \mathcal{P}$, exit($\ell, \mathcal{P}$)))
        else
⟨11⟩        Let $tr$ be $\ell \xrightarrow{\Delta} \ell'$ and $s \xrightarrow{tr} s'$
⟨12⟩        $S'$ := Summarize($s', \mathcal{P}$)
⟨13⟩        $S$ := Compose($s, S_1, S'$)
⟨14⟩        $\overline{S}$ := Join($s, S$, TransStep($s, \mathcal{P}$, exit($\ell, \mathcal{P}$)))
⟨15⟩ else $\overline{S}$ := TransStep($s, \mathcal{P}$, outgoing($\ell, \mathcal{P}$))
⟨16⟩ memo and return $\overline{S}$
```

witness formulas would identically be $\langle(([-,\infty]), F\rangle$. Second, s is a terminal state (line 4). Here Id refers to the identity function, which keep the values of variables unchanged. The ending point of a loop is treated similarly in the third base case (line 5). The last base case, lines 6-7, is the case when a summarization can be reused. We have demonstrated this step at the end of Section 3.

Expanding to the next programming point: Line 15 depicts the case when transitions can be taken from current program point ℓ, and ℓ is not a loop starting point. Here we call TransStep to move recursively to next program points. TransStep implements the traversal of transition steps emanating from ℓ, denoted by

```
function TRANSSTEP($s, \mathcal{P}, TransSet$)
        Let $s$ be $\langle \ell, \cdot \rangle$
⟨17⟩ $\overline{S} := [\ell, T, \langle([+,0]), T\rangle, \langle([+,0]), T\rangle, 0, T, T, Id]$
⟨18⟩ foreach ($tr \in TransSet$) do
⟨19⟩     $s \xrightarrow{tr} s'$
⟨20⟩     $S' :=$ SUMMARIZE($s', \mathcal{P}$)
⟨21⟩     $S :=$ COMPOSE($s$, SUMMARIZE-A-TRANS($s, tr$), $S'$)
⟨22⟩     $\overline{S} :=$ JOIN($s, \overline{S}, S$)
        endfor
⟨23⟩ return $\overline{S}$

function SUMMARIZE-A-TRANS($s, tr$)
        Let $s$ be $\langle \ell, \sigma, \Pi \rangle$ and Let $tr$ be $\ell \xrightarrow{op} \ell'$
⟨24⟩ $\gamma :=$ Sequence of (de-)allocations in $op$
⟨25⟩ Iterate on $\gamma$ and merge consecutive concrete allocations
⟨26⟩ Iterate on $\gamma$ and merge consecutive concrete deallocations
⟨27⟩ $i := 0; netusg := mhw := 0$
⟨28⟩ foreach $[\texttt{sign}, m] \in \gamma$ do
⟨29⟩     if $\texttt{sign}$ $is$ $+$ then $netusg := netusg + m$
⟨30⟩     else $netusg := netusg - m$
⟨31⟩     if $netusg > mhw$ then $mhw := netusg$
        endfor
⟨32⟩ $\Gamma_h := \langle \gamma, [\![op]\!]_\sigma \rangle$;  $\Gamma_n := \langle \gamma, [\![op]\!]_\sigma \rangle$
⟨33⟩ return $[\ell, true, \Gamma_h, \Gamma_n, mhw, true, true, op_\Delta]$

function COMPOSE($s, S_1, S_2$)
        Let $S_1$ be $[\ell_1, \Psi_1, \Gamma_{h1}, \Gamma_{n1}, mhw_1, \delta_{h1}, \delta_{n1}, \Delta_1]$
        Let $S_2$ be $[\ell_2, \Psi_2, \Gamma_{h2}, \Gamma_{n2}, mhw_2, \delta_{h2}, \delta_{n2}, \Delta_2]$
        Let $\Gamma_{n1}$ be $\langle \gamma_{n1}, \pi_{n1} \rangle$
⟨34⟩ $\Delta := \Delta_1 \wedge \Delta_2$
⟨35⟩ $\Psi := \Psi_1 \wedge$ PRE-COND($\Delta_1, \Psi_2$)
⟨36⟩ if ($mhw_1 >$ NET-USG($s, \gamma_{n1}$) $+ mhw_2$)
⟨37⟩     $mhw := mhw_1$
⟨38⟩     $\Gamma_h := \Gamma_{h1}$;  $\delta_h := \delta_{h1}$
       else
⟨39⟩     $mhw :=$ NET-USG($s, \gamma_{n1}$) $+ mhw_2$
⟨40⟩     $\{\Gamma_h, \delta_h\} :=$ COMBINE-WITNESSES($\Delta_1, \Gamma_{n1}, \Gamma_{h2}, \delta_{n1}, \delta_{h2}$)
⟨41⟩ $\{\Gamma_n, \delta_n\} :=$ COMBINE-WITNESSES($\Delta_1, \Gamma_{n1}, \Gamma_{n2}, \delta_{n1}, \delta_{n2}$)
⟨42⟩ return $[\ell_1, \Psi, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta]$

function JOIN($s, S_1, S_2$)
        Let $S_1$ be $[\ell, \Psi_1, \Gamma_{h1}, \Gamma_{n1}, mhw_1, \delta_{h1}, \delta_{n1}, \Delta_1]$
        Let $S_2$ be $[\ell, \Psi_2, \Gamma_{h2}, \Gamma_{n2}, mhw_2, \delta_{h2}, \delta_{n2}, \Delta_2]$
⟨43⟩ $mhw :=$ MAX($mhw_1, mhw_2$)
⟨44⟩ $\Psi := \Psi_1 \wedge \Psi_2$
⟨45⟩ $\Delta := \Delta_1 \vee \Delta_2$
⟨46⟩ $\{\Gamma_h, \delta_h\} :=$ MERGE-WITNESS-H($\Gamma_{h1}, \Gamma_{h2}, \delta_{h1}, \delta_{h2}$)
⟨47⟩ $\{\Gamma_n, \delta_n\} :=$ MERGE-WITNESS-N($s, \Gamma_{n1}, \Gamma_{n2}, \delta_{n1}, \delta_{n2}$)
⟨48⟩ return $[\ell, \Psi, \Gamma_h, \Gamma_n, mhw, \delta_h, \delta_n, \Delta]$
```

Figure 6: Helper Functions

OUTGOING($\ell, \mathcal{P}$), by calling Summarize recursively and then compounds the returned summarizations into a summarization of ℓ. For each tr in $TransSet$, TRANSSTEP extends the current state with the transition. The resulting child state is then given as an argument in a recursive call to SUMMARIZE (line 20). From each summarization of a child returned by the call to SUMMARIZE, the algorithm computes a summarization, contributed by that particular child to the parent as in line 21. Finally, all of these summarizations will be compounded using the Join function (line 22).

SUMMARIZE-A-TRANS computes a summarization for a single transition tr at state s. This can be seen as a basic step in our algorithm. Because no infeasible path has been discovered, the interpolant Ψ is just $true$. There is a single path, thus the dominating conditions are $true$. The abstract transformer Δ is the operation op itself, but translated to the language of input-output relation. As an example, $\mathsf{y} := \mathsf{y} + 1$ is translated to $y_{out} = y_{in} + 1$. We use op_Δ to denote such translated op. We now elaborate on the computation of the witnesses and the high-water mark usage mhw. First, γ is initialized to the sequence of allocations/deallocations, i.e. $[+, e]$ and/or $[-, e]$ in op. Next, consecutive concrete allocation/deallocations are merged by iterating through γ once. Moreover, for each (de-)allocation the $netusg$ is updated in lines 29 and 30. If the value of $netusg$ is greater than the high-water mark value, mhw is updated and ℓ' is stored as the peak location (line 31).

Finally, the path constraint for each witness is computed by projecting op onto the set of program variables w.r.t. the symbolic store σ, denoted as $[\![op]\!]_\sigma$.

Handling Loops: Lines 9-14 handle the case when the current program point ℓ is a loop head. Let ENTRY($\ell, \mathcal{P}$) denote the set of transitions going into the body of the loop, and EXIT($\ell, \mathcal{P}$) denote the set of transitions exiting the loop. Upon encountering a loop, our algorithm attempts to unroll it once by calling the function TRANSSTEP to explore the entry transitions (line 9). When the returned representative path is false, it means that we cannot go into the loop body anymore, we thus proceed to the exit branches. The returned summarization is compounded (using JOIN) with the summarization of the previous unrolling attempt (line 10). Otherwise, if some feasible paths are found by going into the loop body, we first use the returned abstract transformer to produce a new continuation context, (line 11), so that we can continue the analysis with the next iteration (line 12). The returned information is then compounded (lines 13 - 14) with the first unrolling attempt. Our algorithm can be reduced to linear complexity because these compounded summarizations of the inner loop(s) can be reused in later iterations of the outer loop.

Next, we will elaborate on how summarizations are compounded through the helper functions, COMPOSE and JOIN, presented in Figure 6.

Compounding Vertically Two Summarizations: Consider that $subtree(s_2, \ell_3)$ suffixing $subtree(s_1, \ell_2)$, where $s_2 \equiv \langle \ell_2, [\![s_2]\!] \rangle$ and $s_1 \equiv \langle \ell_1, [\![s_1]\!] \rangle$. In other words, a path π_1 from ℓ_1 to ℓ_2 followed by a path π_2 from ℓ_2 to ℓ_3 corresponds a path π in $subtree(s_1, \ell_3)$. The COMPOSE function returns a summarization for $subtree(s_1, \ell_3)$ by compounding the two existing summarizations, respectively for $subtree(s_1, \ell_2)$ and $subtree(s_2, \ell_3)$.

The abstract transformer Δ is computed as the conjunction of the input abstract transformers (line 34), with proper variable renaming. Note that in our implementation, abstract transformers are computed using polyhedral domain. We employ Δ to generate *one* continuation context, before proceeding the analysis with subsequent program fragments. Next, the desired interpolant must capture the infeasibility of S_1, as well as the infeasibility of S_2 given that we treat $subtree(s_1, \ell_2)$ as an abstract transition, of which the operation is Δ. We rely on the function PRE-COND, which in line 35 under-approximates the weakest-precondition of the post-condition Ψ_2 w.r.t. to the transition relation Δ.

Next we update the high-water mark witness. Here the net-usage witness becomes important. In the combined subtree, the high-water mark is chosen by comparing (1) the high-water mark of the prefix tree and (2) the (worst) net-usage of the prefix subtree plus the high-water mark of the suffix subtree (line 36). In case (1) is greater, the witness and the dominating condition of the prefix subtree is returned (lines 37 and 38). Otherwise, the net-usage witness and its dominating condition of the prefix subtree are combined with the high-water mark witness and the corresponding dominating condition of the suffix subtree (lines 39 and 40). This is achieved by calling the function COMBINE-WITNESSES. Finally, we again invoke Combine-Witnesses to combine the net-usage witnesses and their respective dominating conditions (line 41).

```
function Combine-Witnesses($\Delta, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
    Let $\Gamma_1$ be $\langle \gamma_1, \pi_1 \rangle$ and Let $\Gamma_2$ be $\langle \gamma_2, \pi_2 \rangle$
⟨49⟩ $\pi := \pi_1 \ \wedge \ \pi_2$
⟨50⟩ $\delta_2' := true$
⟨51⟩ foreach $cond \in \delta_2$ do
⟨52⟩     $\delta_2' := \delta_2' \ \wedge$ Pre-Cond($\Delta, cond$)
     endfor
⟨53⟩ $\delta := \delta_1 \ \wedge \ \delta_2'$
⟨54⟩ $\gamma_2' := [\,]$
⟨55⟩ foreach $[\texttt{sign}, m] \in \gamma_2$ do
⟨56⟩     $\gamma_2' :=$ Add $[\texttt{sign},$ Pre-Cond$(\Delta, m)]$ into $\gamma_2'$
     endfor
⟨57⟩ $\gamma := \gamma_1 \ \bullet \ \gamma_2'$     // concatenation
⟨58⟩ return $\{\langle \gamma, \pi \rangle, \delta\}$
```

Figure 7: Combining Witness Formula and Dominating Conditions

In Figure 7, Combine-Witnesses produces a witness and a dominating condition, by compounding the witnesses and dominating conditions of the two subtrees, where one suffixes the other. This can be understood as a *sequential* composition.

First, the path constraint π is simply the conjunction of π_1 and π_2 (line 49). Next, the combined dominating condition δ is computed as the conjunction of δ_1 and a condition δ_2', in line 53, where intuitively, δ_2' describes a set of conditions, such that δ_2' is a precondition of δ_2 w.r.t. to the transition relation Δ. Similarly, the allocation/deallocations in γ_2 are updated w.r.t. to the transition relation Δ and stored in γ_2'.

Compounding Horizontally Two Summarizations: Given two summarizations of two subtrees rooted at two nodes which are siblings, we want to propagate the information back and compute the summarization for the (common) parent node. While propagation can be achieved by Compose, we need Join (presented in Figure 6) to "merge" the contributions of the two children to the parent node. Note that unlike Compose, we need to select the path with the larger memory high-water mark usage between the two witnesses of the input summarizations. Thus, the high-water mark usage would be the maximum of the mhw_1 and mhw_2 (line 43). Moreover, all the infeasible paths in both sub-structures must be maintained, thus the desired interpolant is the conjunction of the two input interpolants (line 44). On the other hand, the abstract transformer Δ is computed straightforwardly as the disjunction of the input abstract transformers (line 45). Finally, Merge-Witnesses-h is invoked to merge the high-water mark witnesses and the respective dominating conditions (line 46) and similarly Merge-Witnesses-n is invoked to merge the net-usage witnesses and the respective dominating conditions (line 47).

```
function Merge-Witness-h($\Gamma_1, \Gamma_2, mhw_1, mhw_2, \delta_1, \delta_2$)
⟨59⟩ $\delta := \delta_1 \ \wedge \ \delta_2$
⟨60⟩ if $(true \models mhw_1 \geq mhw_2)$ then return $\{\Gamma_1, \delta\}$
⟨61⟩ if $(true \models mhw_1 \leq mhw_2)$ then return $\{\Gamma_2, \delta\}$
⟨62⟩ return $\{\langle \text{max}(\gamma_1, \gamma_2), \pi_1 \wedge \pi_2 \rangle, \delta\}$

function Merge-Witness-n($s, \Gamma_1, \Gamma_2, \delta_1, \delta_2$)
    Let $\Gamma_1$ be $\langle \gamma_1, \pi_1 \rangle$ and Let $\Gamma_2$ be $\langle \gamma_2, \pi_2 \rangle$
⟨63⟩ $\delta := \delta_1 \ \wedge \ \delta_2$
⟨64⟩ if $(true \models \text{net-usg}(s, \gamma_1) \geq \text{net-usg}(s, \gamma_2))$
⟨65⟩     return $\{\Gamma_1, \delta\}$
⟨66⟩ if $(true \models \text{net-usg}(s, \gamma_1) \leq \text{net-usg}(s, \gamma_2))$
⟨67⟩     return $\{\Gamma_2, \delta\}$
⟨68⟩ return $\{\langle \text{max}(\gamma_1, \gamma_2), \pi_1 \wedge \pi_2 \rangle, \delta\}$
```

Figure 8: Merging Witness Formulas and Dominating Conditions

In Figure 8, Merge-Witnesses-h produces a high-water mark witness and a dominating condition, by compounding the witnesses and dominating conditions of two sibling subtrees. We need to choose one witness from the two input witnesses. The combined dominating condition must ensure the dominance of each witness (in its respective subtree) and the dominance of the chosen witness, which produces a higher value of r, over the other.

The dominating condition δ is initialized as the conjunction of the two dominating conditions (line 59). Next we compare the two high-water marks mhw_1 and mhw_2; if mhw_1 is greater or equal to mhw_2, Γ_1 dominates Γ_2, and Γ_1 is returned as the dominating witness with δ as the dominating condition (line 60). If not, we check if mhw_2 is greater or equal to mhw_1 and then we return Γ_2 as the dominating witness with δ as the dominating condition (line 61). If both tests fail, this could happen when we deal with symbolic expressions, we then employ the max function, delaying the dominance test to a higher level in the symbolic execution tree (line 62) with the hope that another witness might dominate this path. Similarly, the Merge-Witnesses-n function produces a net-usage witness and a dominating condition. Here we make use of the function net-usg, which extracts the net usage given a context (e.g. s) and a sequence of allocations/deallocations (e.g. γ_1 or γ_2). This function can be easily implemented, we omit the detail due to space reason.

Theorem 1 (Soundness)**.** *Our algorithm always produces* safe *worst-case heap memory consumption estimates.*

6. Experimental Evaluation

We evaluate our proposed algorithm using a number of benchmarks collected from the literature. The suite includes: (1) memory allocator tests such as `shbench`, `larson` and `cache-scratch` from Hoard benchmarks [15]; (2) embedded programs from MiBench [17] and from Mälardalen WCET benchmarks [25]); and (3) heap manipulating benchmarks from [23].

In Table 1, the third column indicates the values to which input parameters are concretized. $Time$, $States$ and $Reuses$ columns present the running time, the number of visited states, and the number of reuses in each benchmark instance. In other words, they illustrate the cost of our algorithm.

Among the benchmarks, `nsichneu`, `statemate` and `ndes` contain many (possibly infeasible) paths. They are to stress the scalability of our algorithm. Benchmarks `cache-thrash` and `cache-scratch` are used to test active and passive false sharing. To test for memory fragmentation, `shbench` is often used. It randomly allocates and deallocates random memory chunks of memory. As illustrated in Section 4, our analysis can generate bounds even for programs where memory allocations/deallocations are highly randomized. Finally, `larson` is a famous benchmark which simulates a server. Similar to `shbench` it has a random behavior in memory allocation/deallocation. The analyzed benchmarks are categorized into four groups, separated by a double line in Table 1. We discuss each individual group as below.

The first group of benchmarks contain complicated patterns of allocations/deallocations, e.g. inside loops and conditional branches. In these benchmarks, path-sensitivity plays a crucial role in generating a precise worst-case estimates. Although the method presented in [10] also benefits from path-sensitivity, one key distinction of this work is the employment of symbolic witnesses, which make our analysis applicable to programs with unbounded allocations/deallocations.

Moreover, let us elaborate on `puzzle` to highlight the impact of addressing the issue of non-cumulative resource *directly*, as op-

Table 1: Analysis of Experimental Benchmarks

Benchmark	LOC	Input Parameters (Concretized)	Time	States	Reuses	Bound
larson	614	threads = 1, num_chunks = 100	55.55	613	198	240050000
ndes	219	N.A.	21.21	643	201	11214
puzzle	197	N.A.	164.43	1094	354	204
fasta	121	N.A.	0.23	91	17	755
chomp	401	N.A.	1.59	153	36	6800
cache-thrash	120	threads = 1, iterations = 100, objSize = 1, repetitions = 10	7.96	344	108	1
cache-thrash	120	threads = 2, iterations = 100, objSize = 1, repetitions = 10	8.00	329	103	1
cache-thrash	120	threads = 1, iterations = 200, objSize = 1, repetitions = 10	58.96	644	208	1
cache-thrash	120	threads = 2, iterations = 200, objSize = 1, repetitions = 10	57.60	629	203	1
cache-scratch	126	threads = 1, iterations = 100, objSize = 1, repetitions = 10	9.32	350	108	9
cache-scratch	126	threads = 2, iterations = 100, objSize = 1, repetitions = 10	9.26	338	104	18
cache-scratch	126	threads = 1, iterations = 200, objSize = 1, repetitions = 10	68.86	650	208	9
cache-scratch	126	threads = 2, iterations = 200, objSize = 1, repetitions = 10	69.40	638	204	18
statemate	1090	N.A.	233.63	3553	1296	64
nsicheneu	3144	N.A.	791.75	3639	1376	112
shbench	121	threads = 1, Nalloc = 100	554.39	4461	1408	43600
himenobmtxpa	272	N.A.	175.45	1472	406	57344
dry	491	N.A.	0.3	142	39	112
fft1 (main)	234	MAXWAVES = 8	5.23	88	23	16 * MAXSIZE + 64
nsieve-bits	33	N.A.	4.74	552	192	sz / 8 + 4
ffbench	287	Asize = 10	138.56	1550	508	262160

posed to applying some known approaches for analyzing cumulative resource. In `puzzle`, the outer loop iterates 5 times, and in each iteration it acquires memory, performs some operations in some inner loops and releases the memory at the end of each outer loop iteration. Analyzing memory as a cumulative resource would return 1020 as an estimate of the worst-case consumption, which is 5 times larger than the bound produced by our method.

The second group of benchmarks contains `cache-thrash` and `cache-scratch` which are analyzed for different input parameters. Both `cache-thrash` and `cache-scratch` are multi-threaded benchmarks. We have analyzed their local computation for the cases when number of threads ($nthreads$) are either 1 or 2. In both benchmarks, the number of the iterations of an inner loop is determined by $repetitions/nthreads$. As a result, the execution with $nthreads = 2$ is indeed shorter than the execution with $nthreads = 1$. Finally, note that the generated bound for `cache-scratch` is dependent on the number of threads.

The third group of benchmarks contains `nsichneu` and `statemate`. These two benchmarks contain very large loops which iterate twice. These benchmarks are often used by WCET research community to test the scalability aspect of an algorithm, especially when it is based upon symbolic execution. Our algorithm is able to fully analyze these benchmarks, demonstrating the potential of our new concept of “reuse”.

Finally, the last group of benchmarks contain loops with large number of iterations, where the ability to reuse compounded summarizations to avoid state explosion is crucial. For example, `himenobmtxpa` contains 14 loops with the nested level of 3 and `shbench` contains a complicated loop pattern with the nested level of 3. Among these benchmarks, `shbench` can be executed in both single-threaded and multi-threaded forms which we have analyzed it in the single threaded form. We highlight that the bounds generated for `fft1` and `nsieve-bits` are symbolic bounds.

7. Related Work

We will briefly review three groups of related work.

7.1 Instrumentation Tools

Several different dynamic analysis tools have been developed which perform different forms of memory analysis; they can be categorized under *instrumentation* tools. Such tools often start by profiling an input program before analyzing the collected data; depending on the granularity of the data and the analysis overhead, we can broadly classify into “lightweight” and “heavyweight”.

Firstly, Valgrind [28], is a tool that has been widely used for memory debugging. One of its components, Massif, is a heap profiler that can measure the heap usage in a current execution of the program. Secondly, DynamoRio [7], has a memory debugging tool which can be used to detect heap-overflow errors. One state-of-the-art tool from IBM, Pin [24], tracks the amount of system resources used by a program. Finally, WMTrace [29], is most relevant to this paper. It tracks memory allocation events in a multi-threaded program and in a post processing phase, it measures the worst-case heap usage of the program. All these methods are based on dynamic analysis, and not able to calculate the worst-case memory consumption.

7.2 Worst-case Stack Consumption

Worst-case stack analysis is important for detecting stack overflows in safety-critical embedded systems. One state-of-the-art tool is AbsInt’s StackAnalyzer [21]. It is a variant of values analysis performed on memory cells and CPU registers where the highest value on stack pointer is reported as the worst-case stack usage. This approach employs interval analysis and for precision, it is context-sensitive. Contexts are differentiated by a call string, which is bounded to some N (for scalability). The value analysis keeps updating the intervals till a fixpoint is reached. The highest value on the stack pointer shows the worst-case stack usage.

Recently, [8] employs a variant of Hoare logic to establish bounds on stack usage of C programs. However, this method cannot be extended for dynamic heap allocation. In its current formulation, it requires the size of the stack frame to be static.

In contrast, our method can be used for analyzing both worst-case heap and stack consumption. Importantly, our approach is path-sensitive, thus it produces more precise results. The benefits of path-sensitivity, via symbolic execution, have been argued and also evidenced in many recent works (e.g. [13]).

7.3 Worst-case Heap Consumption

Recently, object oriented languages have been proposed to be used in real-time critical systems [5, 31]. In such languages, besides WCET analysis, analysis of worst-case heap consumption is also crucial for ensuring safety of the deployed systems [30].

One attempt is [30], targeting Java. It employs IPET-based framework, originated from WCET analysis [22]. The method does not take into account memory deallocations and thus the bounds it produces would be imprecise. Another similar work is [4], which only measures the allocations and assumes scope-based memory model: all the allocations are deallocated with the entire scope.

Our most closely related work is the static analysis presented in [10], where an algorithm, extended from [11, 12], has been outlined to perform non-cumulative resource analysis. This algorithm, however, assumes that the amount of resource consumed by each basic block is a constant. In this paper, we make no such assumption. In fact, to accommodate that, we need to introduce a new form of reuse, as detailed in Section 3.

Last but not least, inferring *parametric* bounds, for memory consumption of imperative and object-oriented programs, has been an important research topic [1, 2, 6, 9, 16, 18, 19]. We have carefully discussed their representatives in previous Sections.

8. Conclusion

In this paper, we have presented an algorithm – based on symbolic execution, for analyzing program worst-case memory consumption. In order to adapt for analysis of memory, a cumulative resource, we have modified the concept of "reuse"; now the reused witnesses can also be in the form of a symbolic expression. Finally, we have demonstrated the potential of our approach by showing that our algorithm scales to some realistic benchmarks.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP*, pages 157–172. Springer, 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[3] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quirós, A. Bartzas, F. Catthoor, and D. Soudris. Dynamic memory management optimization for multimedia applications. In *Dynamic Memory Management for Embedded Systems*, pages 167–192. Springer, 2015.

[4] J. L. Andersen, M. Todberg, A. E. Dalsgaard, and R. R. Hansen. Worst-case memory consumption analysis for scj. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 2–10. ACM, 2013.

[5] T. Bøgholm, C. Frost, R. R. Hansen, C. S. Jensen, K. S. Luckow, A. P. Ravn, H. Søndergaard, and B. Thomsen. Towards harnessing theories through tool support for hard real-time java programming. *Innov. Syst. Softw. Eng.*, 9(1):17–28, March 2013.

[6] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM*, pages 141–150. ACM, 2008.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275. IEEE, 2003.

[8] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for c programs. In *PLDI*, pages 270–281. ACM, 2014.

[9] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *PLDI*, pages 467–478. ACM, 2015.

[10] D.-H. Chu. *Interpolation Methods for Symbolic Execution*. PhD thesis, NATIONAL UNIVERSITY OF SINGAPORE, 2012.

[11] D.-H. Chu and J. Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *EMSOFT*, pages 319–328. ACM, 2011.

[12] D.-H. Chu and J. Jaffar. Path-sensitive resource analysis compliant with assertions. In *EMSOFT*, pages 1–10. IEEE, 2013.

[13] D.-H. Chu, J. Jaffar, and R. Maghareh. Precise cache timing analysis via symbolic execution. In *RTAS*, pages 293–304. IEEE, 2016.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[15] D. B. Emery, S. M. Kathryn, D. B. Robert, and R. W. Paul. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS*, pages 117–128. ACM, 2000.

[16] A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In *APLAS*, pages 275–295. Springer, 2014.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization*, pages 3–14. IEEE, 2001.

[18] R. Haemmerlé, P. López-García, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo. A transformational approach to parametric accumulated-cost static profiling. In *FLOPS*, pages 163–180. Springer, 2016.

[19] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS*, 34(3):14:1–14:62, November 2012.

[20] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, pages 297–303. AAAI Press, 2008.

[21] D. Kästner and C. Ferdinand. Proving the absence of stack overflows. In *SAFECOMP*, pages 202–213. Springer, 2014.

[22] Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, 1995.

[23] Llvm test suite guide. URL http://llvm.org/releases/2.2/docs/TestingGuide.html, 2015.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200. ACM, 2005.

[25] Mälardalen WCET research group benchmarks. URL `http-p://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.

[26] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo. A constant-time dynamic storage allocator for real-time systems. *Real-Time Syst.*, 40(2):149–179, November 2008.

[27] M. Masmano, I. Ripoll, and A. Crespo. Dynamic storage allocation for real-time embedded systems. In *RTSS, Work In Progress*, 2003.

[28] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100. ACM, 2007.

[29] O. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Wmtrace – a lightweight memory allocation tracker and analysis framework. In *Proceedings of the UK Performance Engineering Workshop*, 2011.

[30] W. Puffitsch, B. Huber, and M. Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation*, pages 464–478. Springer, 2010.

[31] M. Schoeberl. Scala for real-time systems? In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 1–5. ACM, 2015.

[32] P. W. Trinder, M. I. Cole, K. Hammond, H.-W. Loidl, and G. J. Michaelson. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience*, 25(3):309–348, 2013.