Static Bound Analysis of Dynamically Allocated Resources for C Programs

Guangsheng Fan[†‡] Taoqing Chen[†‡] Banghu Yin[§✉] Liqian Chen[†✉] Tengbin Wang[†] Ji Wang[†‡]
[†]College of Computer, National University of Defense Technology, Changsha, China
[‡]HPCL, National University of Defense Technology, Changsha, China
[§]College of System Engineering, National University of Defense Technology, Changsha, China
{guangshengfan, chentaoqing, bhyin, lqchen, tengbin.wang, wj}@nudt.edu.cn

***Abstract*—It is widely desired to precisely predict bounds of resource usages statically in a program, particularly when the program runs in resource-limited contexts. The resource bound problem becomes more challenging for C programs due to the allowed flexible manipulations on dynamically allocated resources in C. In this paper, we present a static analysis approach to deriving the bounds of dynamically allocated resources for C programs. The key idea is to combine numerical value analysis with pointer analysis under the unified framework of abstract interpretation. First, to track resource usage, we introduce auxiliary numerical variables to model the resource usage due to resource-manipulating functions such as allocation and deallocation. Second, to handle resource-manipulating functions involving pointers as parameters or return values, we propose a pointer analysis approach designed specifically for resource bound analysis, and combine it with numerical value analysis, to handle pointer arithmetics, dynamic allocation and deallocation, etc. Then, we infer the value bound of auxiliary resource-usage modeling variables to predict resource bounds at each program location. We have implemented our approach in a tool called DARB and conducted experiments on a set of benchmarks extracted from real-world programs. The results show that DARB can deal with C programs with complex resource manipulations.**

***Index Terms*—Static Analysis, Resource Bound Analysis, Abstract Interpretation, Pointer Analysis, Numerical Value Analysis**

I. Introduction

Resources refer to any abstractions that system calls offer to a process, apart from the process itself. Such abstractions liberate programmers from having to know details of the underlying hardware and allow them to write portable code. We call a kind of resources as *dynamically allocated resources*, if they are dynamically allocated and released by function calls explicitly. Dynamically allocated resources account a large percent of resources in practice, including heap memory, sockets, file handles, threads, database connections, etc.

When designing software running in the resource-limited contexts (such as embedded software), developers often need to predicate the worst-case resource bounds of a program statically due to limited amount of hardware resources. When the resources requested by the program exceed the amount of available resources that the hardware can offer, the performance of the program will significantly degrade. In this case, resource bounds would provide a proper guidance in the design and configuration of software. Meanwhile, resource bounds are beneficial to detect vulnerabilities (such as buffer overflow, memory leak, etc.), and help preventing DDoS and side channel attacks.

Lots of research has drawn attention to automatically derive resource bounds for imperative programs in recent years [1]–[11]. Many of them focus on deriving the upper-bound number of visits to a given control location or simply the bound of iterations of a loop (or recursion) for purely numerical programs. In fact, when we look into real-world programs, resource-manipulating operations often involve pointers, size-related parameters, etc. Hence, inferring the number of visits to a program location is only *one aspect* of resource bound analysis of real-world programs, while determining the resource size handled in each resource-manipulating operation is *another aspect*. So far, only few work consider the second aspect. C^4B [5] has considered this, and allows users to use *tick(n)* to specify the amount of resource usage (supporting both increasing and decreasing) operated at a program location. However, the resource-usage size *n* needs to be a constant or symbolic parameter (fixed but arbitrary constant) and its value is specified by users manually.

In practice, the resource-usage size of allocation functions may depend on parameters or program variables. Even for the same type of resources, the size for each allocated resource may be different. Meanwhile, the starting memory address of the dynamically allocated resource is often referred by a pointer which may be propagated to other pointers (alias) in the subsequent statements. Hence, it is challenging to determine the usage size of dynamically allocated resources, particularly to determine statically the size of the resource being released when the parameter of the release function involves pointers.

In this paper, we present a novel approach to automatically derive bounds of dynamically allocated resources for C programs. First, we introduce auxiliary integer variables to track the resource usage, and the values of these auxiliary variables reveal resource bounds of the program. In other words, we transform the resource bound problem into the value bound problem. However, the values of these auxiliary resource-usage modeling variables often depend on the parameters

[*]Banghu Yin and Liqian Chen are both corresponding authors.

of resource-manipulating operations. These parameters may be numerical expressions over program variables or pointers (referring to certain resources). For parameters that are pointers, which is often the case for release operations, we need pointer analysis technique to handle pointers. To this end, second, we propose a pointer analysis specifically designed for resource bound analysis to infer the points-to information of all pointers, and then combine it with numerical value analysis under the unified framework of abstract interpretation to handle pointer arithmetics, dynamical allocation/deallocation operations, etc. Finally, we use numerical value analysis to infer the value bounds for auxiliary resource-usage modeling variables, which reveal resource bound in the program.

We have implemented our approach in a prototype called DARB (Static Analyzer of **D**ynamically **A**llocated **R**esource **B**ounds), and experimentally evaluated its effectiveness by analyzing programs extracted from open-source software like C Producer [12], Redis [13], FastDFS [14], Sod [15], etc. DARB successfully derives the bounds of dynamically allocated resources for these programs, while all the existing tools supporting resource bound analysis (such as LOOPUS [6] [10] [11], C^4B [5], KoAT [1] [4] and Rank [16]) fail to directly handle these programs due to lack of analysis of pointers. In addition, we show that the precision of our analysis can be further improved by leveraging these state-of-the-art resource bound analysis techniques (for numerical programs).

The main contributions of this paper are as follows:

- We propose an automatic static upper bound analysis of dynamically allocated resources for C programs by combining pointer analysis and numerical value analysis under the unified framework of abstract interpretation. For each program location, the analysis can infer both the current resource-usage (resource-occupying) bound and peak resource-usage bound before reaching the current program location.
- We propose a Points-to abstract domain specifically designed for resource bound analysis to infer points-to information, and combine it with numerical abstract domains to infer size information of a dynamically allocated resource (that a pointer points to).
- We implement our resource bound analysis in the tool DARB. Experiments on real-world open-source programs show our approach is promising.

II. Overview

In this section, we give an overview of our approach by illustrating how to analyze the bound of dynamically allocated resources in a motivating example. As shown in Fig. 1(a), the original program first uses two while-loops (Lines 13∼25) to construct and deconstruct a list l_1 headed by $t1$, and then uses another two similar while-loops (Lines 27∼39) to construct and deconstruct list l_2 headed by $t2$. The main differences between l_1 and l_2 are their lengths and the memory size of each node in the list. The lengths of l_1 and l_2 are parameters $n1$ and $n2$ respectively, assuming $n2 \geq n1 \geq 1$. Meanwhile, each node of l_1 and l_2 applies $sizeof(*t) + s1 + 1$ (Line 14) and $sizeof(*t) + s2 + 1$ (Line 28) bytes heap memory resources respectively. Note that the $s1 + 1$ units is to make up the *buf* field, which includes a data content of length $s1$ and ends with 1 terminator.

Note that in this program, there are resource release operations which involve pointers whose values depend on the previous statements manipulating pointers. Moreover, the value of the size of each allocated/deallocated node in different lists needs to be distinguished, since $s1$ and $s2$ are variables and have different values. Due to these factors, existing resource bound analysis methods can not handle this program. We now illustrate our static resource bound analysis step by step.

Tracking Resource Usage. As the first step, we introduce auxiliary integer variables and instrument corresponding statements for resource-manipulating functions (i.e., APIs) into the original program (shown in Fig. 1a) to derive the corresponding resource-usage modeling program as shown in Fig. 1(b). The details of the instrumentation for each kind of resource-manipulating functions will be shown in Table I∼II, and explained in Sec. III(A∼B). Symbol ★ marks statements of global declaration and initialization of auxiliary variables, as well as declaration of our auxiliary function *resc_find_size()* which is used to determine the size of the released resources. Besides, ▲ and ▼ sign operations over auxiliary variables to model allocation and deallocation respectively. This step can be done automatically using a program transformation tool (e.g., Coccinelle [17]). Then we track the usage of dynamically allocated resources in the program using auxiliary variables *resc_heap_cur* and *resc_heap_peak*. The first auxiliary variable tracks the current resource usage (the amount of occupying resources) at the current program location, while the latter tracks the historical peak usage before reaching the current program location. The value bounds of these auxiliary variables represent the resource bounds.

Moreover, for each allocation function called at Line #*l* allocating a resource of size *e* (an arithmetic expression involving program variables), we introduce an auxiliary integer variable *resc_size_alloc_#l* and instrument a statement *resc_size_alloc_#l* = *e*, in order to capture the size of allocated resources and maintain its relation with variables in *e*. Symbol ⋇ marks the instrumented statements of this step in Fig. 1(b).

Analysis of instrumented programs. To obtain the resource bound of resource-usage modeling program in Fig. 1(b), we combine pointer analysis and numerical value analysis in the framework of abstract interpretation. The pointer analysis is conducted mainly based on a Points-to abstract domain $\mathcal{B}^\#$ that records the points-to information of each pointer variable. The numerical value analysis is conducted based on numerical abstract domains $\mathcal{N}^\#$, and we use the Polyhedra abstract domain [18] to illustrate. Now, we detail the computation process of the value bounds of *resc_heap_cur* and *resc_heap_peak* as follows.

After Line 12, we get the initial points-to and numerical

```
1 typedef struct node{
2   struct node *next; char buf[];
3 } Node;
4 //precondition: assume(n2>=n1 && n1>=1);
5 void list_mf(unsigned int n1, unsigned int n2){
6   int i=0;
7   int j=0;
8   int s1=10;
9   int s2=20;
10  Node *t1=NULL, *t2=NULL;
11  Node *curr1=NULL, *curr2=NULL;
12
13  while(i<n1){
14    t1 = malloc(sizeof(*t1)+s1+1);
15    t1->next = curr1;
16    curr1 = t1;
17    i++;
18  }
19
20  while(i>0){
21    t1 = curr1->next;
22    free(curr1);
23    curr1 = t1;
24    i--;
25  }
26
27  while(j<n2){
28    t2 = malloc(sizeof(*t2)+s2+1);
29    t2->next = curr2;
30    curr2 = t2;
31    j++;
32  }
33
34  while(j>0){
35    t2 = curr2->next;
36    free(curr2);
37    curr2 = t2;
38    j--;
39  }
40  return;
41 }
```

(a) Original program

```
1 unsigned int resc_size_alloc_14 = 0;※
2 unsigned int resc_size_alloc_28 = 0;※
3 int resc_heap_cur = 0;★
4 int resc_heap_peak = 0;★
5 extern int resc_find_size(void *p);★
6 typedef struct node{
7   struct node *next; char buf[];} Node;
8 //precondition: assume(n2>=n1 && n1>=1);
9 void list_mf(unsigned int n1, unsigned int n2){
10  int i=0, j=0, s1=10, s2=20;
11  Node *t1=NULL, *t2=NULL;
12  Node *curr1=NULL, *curr2=NULL;
13  while(i<n1){
14    t1 = malloc(sizeof(*t1)+s1+1);//if(!t1) abort();
15    resc_size_alloc_14 = sizeof(*t1)+s1+1;※
16    resc_heap_cur += sizeof(*t1)+s1+1;▲
17    t1->next = curr1; curr1 = t1; i++;
18  }
19  if(resc_heap_cur > resc_heap_peak)▲
20    resc_heap_peak = resc_heap_cur;▲
21  while(i>0){
22    t1 = curr1->next;
23    free(curr1);//if(!curr1) abort();
24    resc_heap_cur -= resc_find_size(curr1);▼
25    curr1 = t1; i--;
26  }
27  while(j<n2){
28    t2 = malloc(sizeof(*t2)+s2+1);//if(!t2) abort();
29    resc_size_alloc_28 = sizeof(*t2)+s2+1;※
30    resc_heap_cur += sizeof(*t2)+l2+1;▲
31    t2->next = curr2; curr2 = t2; j++;
32  }
33  if(resc_heap_cur > resc_heap_peak)▲
34    resc_heap_peak = resc_heap_cur;▲
35  while(j>0){
36    t2 = curr2->next;
37    free(curr2);//if(!curr2) abort();
38    resc_heap_cur -= resc_find_size(curr2);▼
39    curr2 = t2; j--;
40  }
41  return;
42 }
```

(b) Resource-usage modeling program

Fig. 1. Motivating example

abstract state:

$$\mathcal{B}^{\#} : B^{\#}_{t1} = NULL \ \wedge \ B^{\#}_{curr1} = NULL \ \wedge \ \cdots$$
$$\mathcal{N}^{\#} : resc\_heap\_cur = resc\_heap\_peak = 0 \ \wedge \ \cdots$$

When the first time we meet the *malloc* function (i.e, Line 14), we use an abstract (symbolic) address *&alloc_14* to represent the memory block address allocated by the system. The resource size $sizeof(*t)+s1+1$ of this memory block is evaluated to 19 through numerical value analysis. Note that in this paper, we assume that all calls to $malloc\,(e)$ will succeed. That is to say, the call of $malloc\,(e)$ will always return a fresh memory of the applied size e as needed. This can be understood as that we add a statement $\{if\,(!t1) \quad abort();\}$ after the *malloc* statement at Line 14. Then, the abstract state after Line 15 is:

$$\mathcal{B}^{\#} : B^{\#}_{t1} = \{\&alloc\_14\} \ \wedge \ B^{\#}_{curr1} \ = NULL \ \wedge \ \cdots$$
$$\mathcal{N}^{\#} : resc\_heap\_cur = resc\_heap\_peak = 0$$
$$\wedge \ resc\_size\_alloc\_14 = 19 \ \wedge \ \cdots$$

As current resource usage has changed, we track this by the statement marked ▲ at Line 16, and update the value of *resc_heap_cur*. At Line 17, we know the base address of $t1->next$ is $alloc\_14.next$, and update it as the current base address of $curr1$ (i.e.,$NULL$). The abstract state after the first iteration of the first while-loop is:

$$\mathcal{B}^{\#} : B^{\#}_{t1} = B^{\#}_{curr1} = \{\&alloc\_14\}$$
$$\wedge \ B^{\#}_{alloc\_14.next} = NULL \ \wedge \ \cdots$$
$$\mathcal{N}^{\#} : resc\_heap\_cur = resc\_size\_alloc\_14 = 19$$
$$\wedge \ resc\_heap\_peak = 0 \ \wedge \ \cdots$$

After several analysis iterations of the first while-loop, we finally get the convergence of the fixed point iteration, and get the abstract state after the first while-loop:

$$\mathcal{B}^{\#} : B^{\#}_{t1}, B^{\#}_{curr1}, B^{\#}_{alloc\_14.next} = \{\&alloc\_14, NULL\} \wedge \cdots$$
$$\mathcal{N}^{\#} : resc\_heap\_cur = 19 * n1 \ \wedge \ resc\_heap\_peak = 0$$
$$\wedge \ resc\_size\_alloc\_14 = 19 \ \wedge \ \cdots$$

The if statements labeled with ▲ (Lines 19∼20) are used to update the historical peak resource usage, since current resource usages have been changed inside the first while-loop.

By doing this update, the abstract state becomes:

$$\mathcal{B}^{\#} : B^{\#}_{t1}, B^{\#}_{curr1}, B^{\#}_{alloc\_14.next} = \{\&alloc\_14, NULL\} \wedge \cdots$$

$$\mathcal{N}^{\#} : resc\_heap\_cur = resc\_heap\_peak = 19 * n1$$
$$\wedge \ resc\_alloc\_14 = 19 \ \wedge \ \cdots$$

For the second while-loop, the handling of pointer operations is similar to the first while-loop. The main challenge is to determine the resource size deallocated at Line 23. We use an auxiliary function $resc\_find\_size()$ to represent the process of automatically determining resource size released (at Line 24). This process first gets the base address set $\{\&alloc\_14\}$ that the pointer parameter $curr$ points to by querying the Points-to domain. Note that in this paper, we assume that all calls to $free\,(p)$ will succeed (that is to say, p is not $NULL$). Hence, although the analysis infers that the abstract base address of $curr1$ is $\{\&alloc\_14, NULL\}$ before Line 23, the analysis will update the base address of $curr1$ to $\{\&alloc\_14\}$, since we can not deallocate a $NULL$ pointer. This can be understood as that we add a statement $\{if\,(!curr1) \quad abort();\}$ after the free statement at Line 23. Since we have recorded $resc\_size\_alloc\_14 = 19$ as the resource size of base address $\&alloc\_14$ in the numerical abstract state, the resource size that $curr$ points to is determined as 19. Then we get the numerical abstract state after Line 26:

$$\mathcal{N}^{\#} : resc\_heap\_cur = 0 \wedge resc\_heap\_peak = 19{*}n1 \wedge \cdots$$

The analysis process of the subsequent two while-loops is similar as above. Note that each allocated resource size at Line 28 is 29 bytes obtained from numerical value analysis. Thus, after analyzing the third while-loop (Lines 27~32) which constructs list l_2, we get $resc\_heap\_cur = 29 * n2$. We then compare the value of $resc\_heap\_cur$ and $resc\_heap\_peak$ (Lines 33~34). By numerical value analysis, we know $n1 \leq n2 \wedge 19 \leq 29$ and the numerical abstract state after Line 34 becomes:

$$\mathcal{N}^{\#} : resc\_heap\_cur = resc\_heap\_peak = 29 * n2 \ \wedge \ \cdots$$

After analyzing the last while-loop which deallocates list l_2, the final values of resource-usage modeling variables are :

$$\mathcal{N}^{\#} : resc\_heap\_cur = 0 \wedge resc\_heap\_peak = 29{*}n2 \wedge \cdots$$

which means that the peak resource usage in the whole program is $29 * n2$ and the resource-occupying amount at the end of the program is 0. Note that this result is sound and precise, according to the concrete semantics of the program.

III. Tracking Resource Usage

In this section, we present how to transform an original program to a resource-usage modeling program.

A. Modeling Resource Usage

We divide resource-manipulating APIs into two types according to types of change size over the resources, i.e., *constant-size* type and *variant-size* type. While APIs of the first type increase or decrease resource usage amounted to constant size each time, APIs of the latter type increase or decrease resource usage amounted to undetermined size that depends on other parameters (or variables). The modeling of resource usage for these two different types are also different.

TABLE I
Modeling File Handle Resource Usage

Resource operation	**Resource modeling statements**
FILE *fp = fopen(char *filename, int flag);	resc_file_cur += 1; if (resc_file_cur > resc_file_peak) resc_file_peak = resc_file_cur;
fclose(FILE *fp);	resc_file_cur -= 1;

TABLE II
Modeling Heap Resource Usage

Resource operation	**Resource modeling statements**
l: ptr = (void*) malloc(unsigned int size);	resc_size_alloc_#l = size; resc_heap_cur += size; if (resc_heap_cur > resc_heap_peak) resc_heap_peak = resc_heap_cur;
l: ptr = (void*) calloc(unsigned int n, unsigned int size);	resc_size_alloc_#l = n*size; resc_heap_cur += n*size; if (resc_heap_cur > resc_heap_peak) resc_heap_peak = resc_heap_cur;
l: ptr = (void*) realloc(void *old, unsigned int size);	resc_size_alloc_#l = size; int resc_size_pre = resc_find_size(old); int resc_size_delta = size - resc_size_pre; resc_heap_cur += resc_size_delta; if (resc_heap_cur > resc_heap_peak) resc_heap_peak = resc_heap_cur;
free(void *ptr);	resc_heap_cur -= resc_find_size(ptr);

APIs of *constant-size* type, for example, includes *fopen()* and *fclose()* that operate file handle resources, *socket()* and *closesocket()* that operate socket resources, etc. Usually, allocation functions of this type only apply 1 unit size resource from the system each time, and the corresponding release functions release 1 unit size resource. Therefore, modeling resource usages of this type is straightforward. As shown in Table I, we model file handle usages by introducing auxiliary integer variables *resc_file_cur* and *resc_file_peak* in which *file* identifies the resource type, to track the current resource usages and the historical peak usages. Each time we call these APIs, *resc_file_cur* increases or decreases 1, and *resc_file_peak* updates if needed.

For *variant-size* type, the most obvious example APIs are heap-manipulating related functions, including *malloc()*, *calloc()*, *realloc()* and *free()*. The resource usage modeling of heap is shown in Table II. The main differences of the operations over auxiliary variables between constant-size and variant-size types APIs are the determination of the size of the operated resource. Note that we only concentrate on resource size while ignoring the stored contents. According to the concrete semantics, for allocation operations such as *malloc()* and *calloc()*, the allocated sizes are apparent (assuming that all allocations are always successful). For *realloc()*, it is a bit more complex. According to its concrete semantics, we first find out the previous size of the memory block that the parameter pointer points to, using an auxiliary function *resc_find_size()*. The functionality of auxiliary function *resc_find_size(ptr)* is to determine the size of memory that *ptr* points to, and the details of how to analyze the auxiliary function *resc_find_size()*

will be detailed in Section IV-D4. The increase or decrease amount of resource usage of *realloc()* is the increment between the new size and the previous size. The released size of *free* function is also determined by the auxiliary function *resc_find_size()*.

B. *Instrumentation of Auxiliary Variables and Statements*

To automatically deriving resource-usage modeling programs, we need to insert necessary resource-modeling statements (e.g., those presented in Table I∼II) into the original program. To do this process automatically, we can take strength of existing work on program instrumentation and make use of automatic tools, such as Coccinelle [17].

When instrumenting resource-usage modeling statements into the original program, we usually instrument these statements immediately after the resource-manipulating APIs, like Line 24 in Fig. 1(b). This is a natural way to track the resource usage change. Modeling statements for allocation functions normally include adding an *if* branch to update the value of *resc_heap_peak* immediately after the update statements of *resc_heap_cur* (as shown in Table II). However, it is worthy noting that for a loop including only allocation and containing no deallocation, the update for *resc_heap_peak* can be delayed after the end of the loop. Based on this insight, as shown in Fig. 1(b), we delay the instrumentation of the *if* statements that update *resc_heap_peak* to the end of the while-loop (Lines 19∼20, Lines 33∼34), since there are just allocations inside the loop. Note that this optimization does not change the semantics, but can improve efficiency and sometimes even precision of the analysis. For those loops that include both allocation and deallocation operations, we still instrument the peak-update statements immediately after the update statements of *resc_heap_peak*.

IV. Analysis of Instrumented Programs

In this section, we introduce how to derive the value bound of the resource-usage modeling auxiliary numerical variables of the instrumented programs. The main idea is to combine pointer analysis and numerical value analysis under the framework of abstract interpretation.

A. *Memory Model of Pointers*

In this subsection, we present the syntax of pointer programs that we consider and the memory model we use.

$$
\begin{array}{lcl}
\mathit{bexp} & ::= & p == q \quad | \quad p \leq q \quad | \quad p \geq q \\
\mathit{exp} & ::= & m * n \quad | \quad m + n \quad | \quad c \ (c \geq 0) \\
S & ::= & p = \&x \quad | \quad p = q + \mathit{exp} \\
 & | & p = \&A[\mathit{exp}] \quad | \quad p = \mathbf{malloc}(\mathit{exp}) \\
 & | & p = \mathbf{malloc}(\mathit{exp}) \quad | \quad p = \mathbf{calloc}(c, \mathit{exp}) \\
 & | & p = \mathbf{realloc}(q, \mathit{exp}) \quad | \quad p = (\text{void*})\ c \\
 & | & S;S \quad | \quad \text{if } (\mathit{bexp}) \text{ then } S \text{ else } S \\
 & | & \text{while } (\mathit{bexp}) \text{ do } S
\end{array}
$$

Fig. 2. Syntax of pointer programs

A pointer records a memory address during the execution of a program. Consider a simple imperative language shown in Fig. 2. It captures the core pointer operations of C language. The variables *p*, *q* denote pointer variables, while *m*, *n* are integer variables, and *c* represents nonnegative integer constants. A pointer *p* can be assigned to a physical address *c*, the address of one variable x (i.e., &x), the address of an element in array *A* (i.e., &A[exp]), or the returned address of an allocation operation. For any other complex assignments to *p*, we can transform them to SSA (Static Single Assignment) form, and derive programs in the form shown in Fig. 2.

In this paper, we use a pair <*baseaddr*, *offset*> as the memory model to describe a pointer at each program location. *baseaddr* means the base address of a memory region, which can be physical address, base address of an array, or access path expression for member field of complex struct type. For a returned pointer *p* of an allocation function, we abstract the returned address of the (dynamically allocated) region as *&alloc_#line*, where *#line* denotes the program location (e.g., the line number) where the allocation function being called. Note that this notion abstracts (over-approximates) the concrete semantic of the allocation function, although in the concrete semantics, different times of calls to the allocation function at *#line* returns addresses of different memory regions. More clearly, *&alloc_#line* can be considered as the set of all possible addresses returned at *#line* through the allocation function. Note that, our analysis for *realloc()* is also sound. The reason is that, in the concrete semantics, the new allocated size may be so large that systems have to return one another large enough memory block to satisfy the need and free *q* after copying contents in *q* to the new block. In our analysis, we just consider that it returns a new address *&alloc_#line* when the realloc function is called at Line *#line*. This is sound, though p may equal to q in the concrete semantics. *offset* is an integer value, and denotes the offset of the current address of a pointer with respect to its base address.

```
1 struct s{
2   int data; int buf[8];}s1;
3 void pointer_model(){
4   int *p = s1.buf;
5   s *q = malloc(sizeof(
        struct s));
6   int *d = &s1.data;
7   int *t=(int*)(0x10000000);
8   int *r = p+2;
9 }
```

Fig. 3. An example pointer program

TABLE III
Memory model

Ptr	Memory model
p	<s1.buf, 0>
q	<&alloc_5, 0>
d	<&s1.data, 0>
t	<0x10000000, 0>
r	<s1.buf, 2>

Example 1. For pointer program in Fig. 3, its corresponding memory model of all pointers in the program are shown in Table III. Note that we assume different variables have different names (after converting to the SSA form).

Note that in the concrete semantics, the possible base address and offset information of a pointer p may be $\{\langle base_1, offset_1\rangle, ..., \langle base_n, offset_n\rangle\}$ at a program location, while in that abstract semantics, we abstract it as $\langle B_p^\#, O_p^\# \rangle$ where $B_p^\# = \{base_1, ..., base_n\}$ and $O_p^\# =$

$$\begin{array}{rcl}
[\![p=\&x]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{\&x\}\right)\wedge[\![\mathcal{O}_p^{\#}=0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=q+exp]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_q^{\#}\right)\wedge[\![\mathcal{O}_p^{\#}=O_q^{\#}+expr]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=\&A\,[exp]]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{A\}\right)\wedge[\![\mathcal{O}_p^{\#}=exp]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=A+exp]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{A\}\right)\wedge[\![\mathcal{O}_p^{\#}=exp]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=(void*)\,c]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{c\}\right)\wedge[\![\mathcal{O}_p^{\#}=0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=malloc\,(exp)]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{\&alloc\_\#l\}\right)\wedge[\![\mathcal{O}_p^{\#}=0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=calloc\,(c,exp)]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{\&alloc\_\#l\}\right)\wedge[\![\mathcal{O}_p^{\#}=0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p=realloc\,(q,exp)]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\{\&alloc\_\#l\}\right)\wedge[\![\mathcal{O}_p^{\#}=0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)
\end{array}$$

Fig. 4. Assignment transfer function

$$\begin{array}{rcl}
[\![p\geq\&x]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{\&x\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq 0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq q+exp]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#},\mathcal{B}_q^{\#}\mapsto\mathcal{B}_{p^{\#}}\cap\mathcal{B}_q^{\#}\right)\wedge[\![\mathcal{O}_p^{\#}\geq O_q^{\#}+exp]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq\&A\,[exp]]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{A\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq exp]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq A+exp]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{A\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq exp]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq(void*)\,c]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{c\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq 0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq malloc\,(exp)]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{\&alloc\_\#l\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq 0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq calloc\,(c,exp)]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{\&alloc\_\#l\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq 0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)\\
[\![p\geq realloc\,(q,exp)]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right) & \triangleq & \left(\mathcal{B}_p'^{\#}\mapsto\mathcal{B}_p^{\#}\cap\{\&alloc\_\#l\}\right)\wedge[\![\mathcal{O}_p^{\#}\geq 0]\!]^{N^{\#}}\left(\mathcal{O}^{\#}\right)
\end{array}$$

Fig. 5. Test transfer function

$offset_1 \cup ... \cup offset_n$, where $\cup$ denotes the join operation in the numerical abstract domain. In other words, in the abstract semantics, we can not distinguish the offset information of different base addresses.

B. Points-to Abstract Domain

For the sake of convenience of combining pointer analysis with numerical value analysis, we design a Points-to abstract domain under the framework of abstract interpretation. To deal with procedures, we inline function calls to get unique program locations of each allocation. Therefore, our pointer analysis is flow-sensitive and context-sensitive. As we consider all possible paths of the fields of each structure, the pointer analysis is also field-sensitive.

This Points-to abstract domain only considers the points-to information of all pointers of a program. For each pointer variable p, the domain records a set $\mathcal{B}_p^{\#} = \{base_1, base_2, ..., base_n\}$ of base addresses that the pointer variable p may point to. Thus $\mathcal{B}_p^{\#}$ constitutes a complete lattice $(\mathcal{P}(\mathcal{B}), \subseteq, \cap, \cup, \bot, \top)$, where $\mathcal{B}$ denotes the set of all possible base addresses in the whole program, $\subseteq$, $\cap$ and $\cup$ mean subset inclusion, intersection and union operation of set, while $\bot$ means pointer p does not point to any address (i.e., $\mathcal{B}_p = \varnothing$), and $\top$ is the set of all base address expressions in the program (i.e., $\top = \mathcal{B}$).

Since there are always limited finite numbers of base address expressions in a program, the complete lattice constituted by the set of base address expressions has a finite height. That is to say, we do not need to design widening and narrowing operation for this abstract domain, to guarantee the convergence of fixpoint iterations. Other domain operations such as assignment and test transfer functions will be detailed in Section IV-D.

C. Numerical Abstract Domain

Numerical abstract domains are used to infer automatically the numerical relations among program variables of numerical type in a program. E.g., the Polyhedra abstract domain is used to infer numerical invariants of the form $\sum_{i=0}^{n} a_i x_i \leq c$, where x_i's are program variables, and coefficients a_i's as well as c are inferred automatically by the analysis. A numerical abstract domain provides domain operations with efficient algorithms to manipulate domain elements. Common domain operations include inclusion testing $\sqsubseteq$, meet $\sqcap$, join $\sqcup$, widening ∇, narrowing $\triangle$, assignment and test transfer functions. Widening and narrowing operators are designed to accelerate and guarantee the convergence of iterations of fix-point computation. We refer the details of the design of numerical abstract domain to tutorial [19].

Particularly, in our resource bound analysis, the numerical abstract environment consists of three kinds of numerical variables: 1) program variables of numerical type in the original program; 2) instrumented auxiliary integer variable for modeling resource usage; 3) numerical variables $O_p^{\#}$ capturing the offset of pointer p. We use numerical abstract domains to infer the numerical relations among these three kinds of numerical variables.

D. Resource Bound Analysis by Combining Points-to and Numerical abstract Domains

The key idea of this paper is to conduct resource bound analysis by combining Points-to and numerical abstract domains. Since statements containing only numerical variables will be handled automatically by conventional numerical value analysis using numerical abstract domains, in the following, we only describe how to analyze statements involving pointer variables.

1) Assignment Transfer Function: Assignment transfer functions reveal transfer operations on abstract states once we meet an assignment statement. For pointer assignments listed in Fig. 2, we design their corresponding abstract assignment transfer functions as shown in Fig. 4. $[\![assignment]\!]^{P\#}$ is the abstract transfer function in the combined domain for $assignment$, and its input $(\mathcal{B}^{\#}, \mathcal{O}^{\#}, l)$ consists of abstract environment of base address $\mathcal{B}^{\#}$ before executing the *assignment*, as well as the *offset* $\mathcal{O}^{\#}$ represented in the numerical abstract domain, and the program location l of the assignment. $\mathcal{B}^{\#}_{p}$ and $\mathcal{B}'^{\#}_{p}$ denote abstract base address of *p* before and after the assignment respectively. $[\![assignment]\!]^{N\#}$ is the transfer function of numerical abstract domain $\mathcal{N}^{\#}$. In a word, the assignment transfer function to a pointer *p* not only updates base address $\mathcal{B}^{\#}_{p}$ that *p* points to, but also maps the operation over its *offset* $\mathcal{O}^{\#}_{p}$ to assignment transfer function in the numerical abstract domain.

2) Test Transfer Function: Test transfer functions $[\![expr_1 \bowtie expr_2]\!]^{P\#}$ reveal transfer operations on abstract states once we meet a condition test statement that involves pointers. They aim to filter abstract states that do not satisfy the condition $expr_1 \bowtie expr_2$, where $\bowtie \in \{\neq, =, <, \leq, >, \geq\}$. Note that the test $expr_1 \bowtie expr_2$ is meaningful, only when exp_1 and exp_2 have common base address. In Fig. 5, we just consider test condition in the form of $p \geq expr$, since the other forms can be transferred to this form. Like assignment transfer functions, we also consider updating both abstract base address $\mathcal{B}_p$ and numerical offset $\mathcal{O}_p$ (maintained in the numerical abstract domain) of a pointer *p*.

3) Extrapolations: To handle loops. we need extrapolation operations (widening, narrowing) to accelerate and guarantee the convergence of fixpoint iterations. As mentioned in Section IV-B, the complete lattice constituted by the set of base addresses is of finite height, and thus we do not need widening and narrowing operations for the Points-to abstract domain. However, for the numerical part, i.e., numerical abstract states (over program variables of numerical type, instrumented integer auxiliary variables modeling resource usage, numerical variables $\mathcal{O}^{\#}_{p}$ capturing offsets of pointers), we still need to the widening/narrowing operation from the numerical abstract domain at loop heads.

4) Size Analysis of Released Resources: Now, we introduce how to deal with the auxiliary function *resc_find_size()* which aims to determine the resource size to be released. Remind that for each allocation function at the program location l, we have set $\&alloc\_\#l$ as the base address of the returned pointer variable in the Points-to domain (as shown in Fig. 4), and have instrumented an assignment statement to an auxiliary variable $resc\_size\_alloc\_\#l$ to capture the size of the allocated resources. The numerical relations among variable $resc\_size\_alloc\_\#l$ and other program variables are maintained in the numerical abstract domain. Once we meet the statement calling the auxiliary function *p_size = resc_find_size(p)*, we first query Points-to domain for the base address $\mathcal{B}^{\#}_{p}$ of its argument pointer *p*. After that, from the Points-to domain, we get the set of base addresses that *p* may point to, i.e., $\mathcal{B}^{\#}_{p} = \{base_1, base_2, ..., base_n\}$. Then from $B^{\#}_{p}$, we can derive a subset $\mathcal{B}^{\#}_{p\_alloc} = \{base_1, base_2, ..., base_m\}$, where each $base_i$ $(1 \leq i \leq m)$ is in the form $\&alloc\_\#l_i$. For each $\&alloc\_\#l_i$, we can get $resc\_size\_alloc\_\#l_i$ denoting the corresponding allocated resource size, which is a numerical variable in the numerical abstract environment. Finally, the returned value of *resc_find_size()* is the join of all those $resc\_size\_alloc\_\#l_i$'s, which is a sound upper approximation. In other words, *p_size = resc_find_size(p)* can be interpreted as $\overset{m}{\underset{i=1}{\sqcup}}{}^{N\#} [\![p\_size = resc\_size\_alloc\_\#l_i]\!]^{N\#}$, where $\sqcup^{N\#}$ denotes the join operation in the numerical abstract domain. In this way, we can analyze the resource usage of a release function.

Example 2. In Fig. 1(b), we instrument auxiliary variable $resc\_size\_alloc\_14$ to record the allocated resource size that pointer $t1$ points to at Line 14, and it is straightforward to obtain its value 19 from numerical abstract domain. When we release pointer $curr1$ at Line 23, we first get the set of the base addresses (i.e., $\mathcal{B}^{\#}_{curr1} = \{\&alloc\_14\}$) $curr1$ points to from the Points-to domain. From $\{\&alloc\_14\}$, we know that $curr1$ points to the memory region allocated at Line 14, and the resource size of this region is $resc\_size\_alloc\_14$ (whose value is 19). Then, we know the size of the resource to be released at Line 23 is 19.

E. Supporting pointer arithmetics inside a structure

In the above subsection, we only consider core pointer operations of C. In fact, pointer operations in real-world programs are often much more complex. Programmers may use pointer arithmetics inside a structure to reference different fields. The program shown in Fig. 6 reveals a common way of using pointers. The program uses data structure *sdshdr*, which is an efficient and safe implementation form of string to store data. The construction of *struct sdshdr* through the function *sdsnewlen()* just returns the address of the buf field which is used to store data (as shown at Line 12), rather than the starting address of the whole *struct sdshdr*. The base address of the returned pointer is $\&alloc\_8.buf$ (note that Line 8 is exactly the program location calling the *malloc()* function). As shown at Line 21, the use mode of *struct sdshdr* often takes this returned pointer of *sdsnewlen*() as input, but needs to derive the base address of its structure memory block (i.e., $\&alloc\_8$), in order to operate the other fields of the structure (Line 16). However, using transfer functions listed in Fig. 4, we can just get $\&alloc\_8.buf$ rather than $\&alloc\_8$, as the base address of parameter of *free* function.

To solve this problem, we first record statically the size of all kinds of struct, and the offset $ofield$ of each field it contains from the beginning of the struct in a table *T*. In this way, when analyzing pointer operations $[\![p = q -$

```
1 typedef char* sds
2 struct sdshdr {
3   unsigned int len;
4   unsigned int free;
5   char buf[];};
6 sds sdsnewlen(void *init, size_t initlen) {
7   struct sdshdr *sh
8   sh = malloc(sizeof(struct sdshdr)+initlen+1);
9   ......
10  sh->len = initlen;
11  sh->free = 0;
12  return (char *) sh->buf;
13 }
14 void sdsfree(sds s) {
15  if (s == NULL) return;
16  free(s - sizeof(struct sdshdr));
17 }
18 void foo(char *in, size_t len){
19  sds s = sdsnewlen(in, len);
20  ...
21  sdsfree(s);
22 }
```

Fig. 6. An example extracted from C-Producer/src/sds.c

$exp]\!]^{P\#}\left(\mathcal{B}^{\#},\mathcal{O}^{\#},l\right)$, we first query the value of exp from the numerical abstract domain. If exp is a nonnegative integer, and the address of q in the Points-to domain is $\mathcal{B}^{\#}_{q} = \{base_1.field_1, base_1.field_2, ..., base_n.field_m\}$, then for each $base_i.field_j, (1 \leq i \leq n, 1 \leq j \leq m)$, we query T for the $ofield$ of $field_j$. If numerical abstract domain determines that $ofield$==exp, we know that the base address of *p* is indeed $base_i$. Note that this method is consistent to the layout of struct memory.

Example 3. In Fig. 6, the size of *sdshdr* is 8, the offset of the field *buf* with respect to the beginning of *sdshdr* is also 8. Since the base address of *s* is $\&alloc\_8.buf$, at Line 16 the base address of the released pointer $s - sizeof(struct\ sdshdr)$, i.e., $\&alloc_8.buf - 8$, which is determined as $\&alloc\_8$ finally.

V. Experiments and Evaluation

We have implemented our approach in a prototype named DARB to automatically analyze the bounds of dynamically allocated resources. For each input C program, DARB first instruments the resource-usage modeling auxiliary variables and corresponding statements (as described in Sect. III) into the original program using Coccinelle [17], which is a program matching and transformation tool. Then DARB uses the approach described in Sect. IV to analyze the instrumented programs. The analyzer is implemented based on the frontend CIL [20], numerical abstract domain library Apron [21], and the Fixpoint Solver Library [22]. Here we use a CIL supported inline tool to inline procedures and use the Polyhedra abstract domain to conduct numerical value analysis.

A. Experimental Setup

We conduct experiments to study the following two research questions (RQs).

- **RQ1.** How DARB performs when analyzing bounds of dynamically allocated resources for realistic programs from open-source projects?
- **RQ2.** How DARB performs with and without the help of existing resource bound analysis tool for purely numerical programs?

To address RQ1, we extract a set of programs as shown in Table IV from four open-source projects, i.e., FastDFS [14], Redis [13], C Producer [12] and Sod [15]. FastDFS is a high performance distributed file system, which contains a lot of file handler resource operations. Redis is a well-known database that contains many socket communications between the clients and servers. C Producer is an embedded log producer developed by Aliyun and has been widely used in millions of intelligent devices. Sod is an embedded computer vision and machine learning library. All the programs extracted from C Producer and Sod involve a lot of heap memory operations. Mot_Ex in Table IV includes our motivating example *list_mf* shown in Fig. 1(a) and several variants through changing the ordering of while-loops. The cyclomatic complexity of these functions are from 1 to 22, and 6.6 on average, and more than 85% of them involve invocations to other functions.

To address RQ2, we use 20 programs from benchmark C4B [5], which has complex loop patterns and are challenging to infer the bound of iterations of a loop, especially when only using numerical abstract domains. However, existing state-of-the-art resource bound analysis tools such as Loopus can compute meaningful upper bound numbers of loop iterations for these programs, which can be used to improve the precision of DARB. We use Loopus to conduct experiments to address RQ2, since it is the only open-source tool that can do analysis directly on C programs as far as we know[1]. The version of Loopus we use is [23]. And we use the set of programs from benchmark of C^4B [5] that can be successfully analyzed by Loopus, i.e., 20 program in Table V, to conduct experiments. In order to support the analysis of DARB, the *tick(1)* statement (which means applying one unit size of resource) in these experimented programs is replaced with *void *ptr= malloc(1)*.

All the experiments are carried out on a virtual machine (using VirtualBox), with a guest OS of Ubuntu 14.04 (8GB RAM), host OS of Windows 10, with 16GB RAM and a 3.0 GHz eight-core Intel(R) Core(TM) i7-9700 host CPU.

B. Answers to RQ1

Table IV shows the results of analyzing programs from FastDFS, Redis, C Producer, Sod and Mot_Ex (listed in column "Project") by DARB. The column "Function" shows the functions that we extracted from above datasets as input programs to be analyzed. The column "Line" shows the number of lines of code in each program after inlining. The column "Time(s)" is the time consumption of analyzing each program in seconds, and does not include the time consumption of code instrumentation since Coccinelle takes almost negligible time (1ms to 10ms, 4ms on average). The column "Peak" and "Cur" reveal the size of the peak resource

[1]C^4B [5] can also conduct resource bound analysis on C programs. However, its publicly available version at http://www.cs.yale.edu/homes/qcar/aaa/ encounters "Error: analysis failure" even for example programs provided in the website, during the preparation of this paper.

TABLE IV
EXPERIMENT RESULTS ON DYNAMICALLY ALLOCATED RESOURCES

Project	Function	Line	Time(s)	Peak	Cur
Fast-DFS	gen_files main	334	0.97	1	0
	do_dispatch_binlog _for_threads	1021	10.98	n	[0, n]
	load_file_contents	219	1.94	[0,7]	[0,7]
	fdfs_binlog _compress_func	581	11.34	1	0
	tracker_mem _get_sys_files	372	2.08	[1,4]	[0,1]
	test_delete main	1017	5.42	2	0
	test_upload main	1409	8.36	[7,9]	[4,6]
Redis	anetListen	176	1.24	0	[-1,0]
	anetGenericAccept	139	0.60	1	1
	_anetTcpServer	446	3.62	[0,1]	[0,1]
C Pro-ducer	sdsnewlen	156	1.13	n+9	0
	sdsnewEmpty	138	0.86	n+9	0
	sdsdup	276	2.53	n+9	0
	sdsfree	226	1.76	0	-n-9
	sdsMakeRoomFor	280	3.02	2m+n	0
	sdsRemoveFreeSpace	284	2.08	0	-n-9
	sdsgrowzero	345	3.21	2m-n	0
	sdscatchar	342	4.24	2+n	0
	sdscatlen	1171	10.29	2m+n	0
	log_queue_create	110	0.98	8n+48	8n+48
Sod	BlobPrepareGrow	549	2.07	2n+16	2n+16
	make_network	907	20.98	808n+8	808n+8
Mot_Ex	list_mf	109	0.78	29n2	0
	list_mf_1	111	0.79	19n1+29n2	0
	list_mf_2	111	0.78	19n1+29n2	0
	list_mf_3	109	0.74	29n2	0

usages during the whole execution and the current resource usages when reaching the end of each program respectively. We derive the lower and upper bounds for "Peak" and "Cur" from the generated numerical invariants by the analysis, which constitute a (symbolic) interval. For the derived interval, when the lower bound equals to the upper bound, we simply use an expression to denote the interval.

FastDFS and Redis are used to evaluate DARB's ability of dealing with resource usage of file handlers and sockets respectively, each API of which has constant-size resource usage. Table IV shows that the peak resource usages (column "Peak") of these programs are mostly constant numbers. We have manually checked the source code of these programs, and found that both allocations and deallocations of the same resource usually happen in the same loop body, thus the peak resource usage of the loop is only 1 unit, even when the loop iteration number is parameterized. Actually, DARB has found the exact (i.e., most precise) upper bound of peak resource usage for all of these programs.

Normally, for a correct program (without memory bugs), the resource usage (i.e., column "Cur") should be 0 (or belong to [0,n], where n represents the peak resource usage) at the end of the program. From column "Cur" in Table IV, we found that, 1) for 5 programs (whose "Cur" are 0), DARB has computed the exact current resource usage; 2) for 3 programs (whose column "Cur" belong to [0,n], where n represents the peak resource usage), DARB has returned over-approximate current resource usage due to precision loss of our analysis or abnormal exit statements contained in the program; 3) for programs *test_upload_main* and *anetListen*, which violate the above principle. We have manually checked their source codes, and observed that program *test_upload_main* ("Cur" is [4,6]) forgets to release 6 already allocated file handlers and program *anetListen* ("Cur" is [-1,0]) only has one resource release operation in one branch of an if-else statement, but has no allocating operation before that.

C producer, Sod and Mot_Ex are used to evaluate DARB's ability of dealing with resource usage of heap memory, each API of which has variant-size resource usage. All the extracted programs from them have complex pointer operations. The columns "Peak" and "Cur" for these programs are all numerical or symbolic values rather than intervals, which indicates that our DARB has computed the exact resource usage bounds for them. And we have confirmed the correctness of the resource bounds by manually checking. We observed that programs *sdsfree*, *sdsRemoveFreeSpace* (*log_queue_create*, *BlobPrepareGrow*, *make_network*) only have memory $free$ ($malloc$) operations, but no memory $malloc$ ($free$) operations. Finally, the column "Time" indicates that the time overhead (within 30s) of DARB is acceptable.

C. Answers to RQ2

Table V shows the results of using DARB only (column "DARB"), and the results of using DARB together with Loopus (column "DARB+Loopus"). We get the result of "DARB+Loopus" by first instrumenting the loop bounds computed by Loopus to the corresponding loop conditions of the target programs. Since the loop bounds given by Loopus contain max expressions which cannot be precisely captured by the Polyhedra abstract domain, we decomposed the max expressions into several if-else branches. E.g., instrumenting the upper bound $\max(x-y,0)$ provided by Loopus to original program $\{$*while(b) s;*$\}$, will result in

```
unsigned int i=0;
if(x-y>=0) { while(b and i<=x-y) {s;i++;}}
else       { while(b and i<=0)   {s;i++;}}
```

And then we use our tool DARB to compute the resource bounds of the instrumented programs. The final resulting bounds of "DARB+Loopus" may contain $\max$ ($\min$) for upper (lower) bound, which is derived (more precisely collected) from the invariants at ends of different instrumented if-else branches (due to the instrumenting of the $\max$ bounds given by Loopus). The symbol $+\infty$ means we can not derive finite resource bound. Column "Cmp" denotes the comparison of analysis precision of with and without using Loopus. "=" means using "DARB+Loopus" can not derive more precise resource bound. ">" means "DARB+Loopus" together with Loopus gets more precise results than that of using only DARB. In particular, we use $_{\infty} >_f$ to denote that using "DARB+Loopus" can derive finite upper bound while using only DARB can derive only infinite upper bound. Similarly, $_f >_f$ means both "DARB+Loopus" and "DARB" can infer finite upper bounds, but "DARB+Loopus" infers a tighter one. Symbol $*$ denotes that the analysis result is an exact value (precise), rather than an over-approximated interval (sound).

From Table V, we can see that DARB can only derive finite upper bounds for 7/20 programs, and only 4/7 of the derived results are exact values (precise). This is because widening operation of the Polyhedra domain used in numerical value analysis may cause precision loss when handling loops in these benchmark programs. Using "DARB+Loopus", the number of each kind of $_{\infty}{>}_{f}$, =, $_{f}{>}_{f}$ is 9, 10, 1 respectively. With the help of Loopus, we can derive finite upper bounds for 17/20 programs, more than two times of the results without using Loopus. Moreover, 10/17 of the derived bounds are more precise, among which "DARB+Loopus" successfully infer finite upper bounds for 9 programs for which "DARB" only infer infinite upper bounds. Experimental results show that combining with state-of-the-art resource bound analysis tools for purely numerical programs can improve the precision of our approach. Hence, it is worthy to combine our approach with the existing work on resource bound analysis only for purely numerical programs.

TABLE V
EXPERIMENT RESULTS ON COMBINING DARB AND LOOPUS

Program	DARB		DARB+Loopus		Cmp
	Peak	Time (s)	Peak	Time (s)	
kmp	[0,+∞)	0.31	[0,+∞)	0.53	=
speed_pldi10_ ex1	[0,n-1]	0.04	[0,n-1]	0.17	=
speed_pldi10_ ex3	[0,n-1]	0.04	[0,n-1]	0.19	=
speed_pldi10_ ex4	[0,+∞)	0.01	[0,+∞)	0.21	=
speed_popl10_ fg2_1	[0,+∞)	0.06	[0,max(m+n-x-y, n-x,1)]	1.92	$_{\infty}{>}_{f}$
speed popl10 fg2_2	[0,+∞)	0.06	[0,max(2n-x-z, 2n-2x-1,1)]	1.91	$_{\infty}{>}_{f}$
speed_popl10_ nested_multiple	[0,+∞)	0.09	[0,max(m+ n-x-y,0)]	2.46	$_{\infty}{>}_{f}$
speed_popl10_ simple_single	n	0.03	n	4.12	=
speed_popl10_ simple_single_2	[0,+∞)	0.05	[min(m,n),m+n]	0.16	$_{\infty}{>}_{f}$
speed_popl10_ sequential_single	n	0.07	n	0.52	=
t07	3x+y	0.13	3x+y	0.48	=
t08	[0,+∞)	0.09	[0,max((4z-3y)/3, (2z-y-z)/3,0)]	0.62	$_{\infty}{>}_{f}$
t10	[0,+∞)	0.03	x-y*	0.38	$_{\infty}{>}_{f}$
t11	[0,+∞)	0.04	[0,max(m+n-x-y,1)]	1.58	$_{\infty}{>}_{f}$
t13	[0,+∞)	0.12	[0,+∞)	2.91	=
t19	i+k+51	0.03	i+k+51	0.82	=
t20	[0,x+y-2]	0.05	[0,max(y-x, x-y)]	2.18	$_{f}{>}_{f}$
t27	[0,+∞)	0.10	[0,+∞)	1.92	=
t28	[0,+∞)	0.07	[0,max(1001x -1000y,0)]	1.01	$_{\infty}{>}_{f}$
t47	[0,+∞)	0.03	[0,n+1]	0.13	$_{\infty}{>}_{f}$

D. Threats to Validity

The main threats to the validity of our results come from the following threat categories. The internal threat to validity mainly lies in whether our implementation of DARB correctly captures our approach. We have manually checked the generated invariants at each program location to ensure the correctness of our analysis. The external threat to validity lies in the dataset. Although the scales of our selected programs are not very large, most of them are extracted from real-world software or classic benchmarks, which are representative and can reflect the applicability of our method. The construct threat to validity comes from the assumptions of our approach. We assume that all calls to allocations/deallocations will succeed.

VI. RELATED WORK

Resource bound analysis of imperative programs. There are many tools that can automatically derive loop and recursion bounds for imperative programs, such as SPEED [8] [9], C^4B [5], KoAT [1] [4], PUBS [2] [3], Rank [16], Loopus [11] [6] [10], CoFloCo [7], etc. KoAT, Rank, CoFloCo and Loopus mainly use term-rewriting, cost equation or difference constraints techniques. SPEED also combines instrumentation and abstract interpretation techniques, which instruments an auxiliary counter variable to record the iteration number of a loop, and uses abstract interpretation to generate invariants for instrumented program, including the value bound of the loop counter variable. However, all these work consider purely numerical programs, and focus on deriving the upper-bound number of visits to a given control location, while our work focus on deriving bounds of dynamically allocated resources for C programs which often involve pointer operations.

The most relevant work to our technology is C4B [5], which derives worst-case resource bounds for C programs by combining amortize resource analysis and abstract interpretation. It has considered using auxiliary function *tick(n)* to specify the size change of resource usage, and *n* can be negative to represent resource releases. However, *n* needs to be determined manually by users. And C^4B does not handle pointer operations. Different from C^4B, our work aims to infer bounds of dynamically allocated resources, and capture the size change of resource usages for each resource-manipulating operation automatically, especially when the operation is related to pointers. Besides, we also derive not only worst-case bounds, but also best-case resource bounds (that are the lower bounds of variables, such as *resc_heap_cur* and *resc_heap_peak*).

Resource bound analysis of functional programs. The resource bound analysis techniques of functional programs are mainly based on sized types [24], inductive types [25], term rewriting [26] or amortize analysis [27]–[29] [30]. The RAML system proposed in [29] can derive worst-case polynomial resource bounds for higher-order polymorphic programs, and has ability to deal with large-scale real-world OCaml programs.

Resource bound analysis of probabilistic programs. Recently, researchers have also drawn attention to resource bound analysis of probabilistic programs. Based on amortized resource analysis, Ngo et al. propose a technique of resource bound analysis for probabilistic programs [31]. Wang et al. present a resource bound analysis technique based on semi-algebraic, and can deal with resource release [32].

VII. CONCLUSION

We have presented a novel approach to automatically inferring worst-case bounds of dynamically allocated resources for C programs. It can derive bounds of both historical peak resource usage and current resource usage of each program location. We first introduce auxiliary variables and corresponding operations to model resource usages of each resource-manipulating API in programs. Then, we transfer resource bound problem into value bound problem of these auxiliary variables. Since operations over dynamically allocated resources often involve pointers, we have designed a Points-to abstract domain specifically for resource bound analysis, and combined it with numerical abstract domain to handle pointer arithmetics, dynamical allocations/deallocations, and to infer the size information of a dynamically allocated resource that a pointer points to. We have developed a tool named DARB, and the experimental results have shown its promising ability to infer the bounds of dynamically allocated resources of real-world programs with complex resource manipulations.

Acknowledgment

This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), and the National Natural Science Foundation of China (Nos. 61872445, 62032024, 62102432).

References

[1] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Alternating runtime and size complexity analysis of integer programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 140–155.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Closed-form upper bounds in static cost analysis," *Journal of automated reasoning*, vol. 46, no. 2, pp. 161–203, 2011.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Cost analysis of object-oriented bytecode programs," *Theoretical Computer Science*, vol. 413, no. 1, pp. 142–159, 2012.

[4] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Analyzing runtime and size complexity of integer programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 38, no. 4, pp. 1–50, 2016.

[5] Q. Carbonneaux, J. Hoffmann, and Z. Shao, "Compositional certified resource bounds," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 467–478.

[6] M. Sinn, F. Zuleger, and H. Veith, "A simple and scalable static analysis for bound analysis and amortized complexity analysis," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 745–761.

[7] A. Flores-Montoya and R. Hähnle, "Resource analysis of complex programs with cost equations," in *Asian Symposium on Programming Languages and Systems*. Springer, 2014, pp. 275–295.

[8] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," *ACM Sigplan Notices*, vol. 44, no. 1, pp. 127–139, 2009.

[9] S. Gulwani and F. Zuleger, "The reachability-bound problem," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 292–304.

[10] M. Sinn, F. Zuleger, and H. Veith, "Difference constraints: An adequate abstraction for complexity analysis of imperative programs," in *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2015, pp. 144–151.

[11] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith, "Bound analysis of imperative programs with the size-change abstraction," in *International Static Analysis Symposium*. Springer, 2011, pp. 280–297.

[12] https://github.com/aliyun/aliyun-log-c-sdk.

[13] https://github.com/redis/redis.

[14] https://github.com/happyfish100/fastdfs.

[15] https://github.com/symisc/sod.

[16] C. Alias, A. Darte, P. Feautrier, and L. Gonnord, "Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs," in *International Static Analysis Symposium*. Springer, 2010, pp. 117–133.

[17] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," *Acm sigops operating systems review*, vol. 42, no. 4, pp. 247–260, 2008.

[18] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978, pp. 84–96.

[19] A. Miné, "Tutorial on static inference of numeric invariants by abstract interpretation," *Foundations and Trends in Programming Languages*, vol. 4, no. 3-4, pp. 120–372, 2017.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *International Conference on Compiler Construction*. Springer, 2002, pp. 213–228.

[21] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *International Conference on Computer Aided Verification*, 2009.

[22] http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/index.html.

[23] https://forsyte.at/software/loopus/.

[24] P. B. Vasconcelos, "Space cost analysis using sized types," Ph.D. dissertation, University of St Andrews, 2008.

[25] N. Danner, D. R. Licata, and R. Ramyaa, "Denotational cost semantics for functional languages with inductive types," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 140–151.

[26] M. Avanzini, U. Dal Lago, and G. Moser, "Analysing the complexity of functional programs: higher-order meets first-order," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 152–164.

[27] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 34, no. 3, pp. 1–62, 2012.

[28] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 185–197, 2003.

[29] J. Hoffmann, A. Das, and S.-C. Weng, "Towards automatic resource bound analysis for ocaml," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 359–373.

[30] H. Simoes, P. Vasconcelos, M. Florido, S. Jost, and K. Hammond, "Automatic amortised analysis of dynamic memory allocation for lazy functional programs," in *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, 2012, pp. 165–176.

[31] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann, "Bounded expectations: resource analysis for probabilistic programs," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 496–512, 2018.

[32] P. Wang, H. Fu, A. K. Goharshady, K. Chatterjee, X. Qin, and W. Shi, "Cost analysis of nondeterministic probabilistic programs," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 204–220.