Static Analysis of Resource Usage Bounds for Imperative Programs

Liqian Chen† Taoqing Chen† Guangsheng Fan† Banghu Yin†
† National University of Defense Technology, Changsha, China
{lqchen, chentaoqing, guangshengfan, bhyin}@nudt.edu.cn

2021 28th Asia-Pacific Software Engineering Conference (APSEC) | 978-1-6654-3784-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/APSEC53868.2021.00077

***Abstract*—Analyzing worst-case resource usage of a program is a difficult but important problem. Existing static bound analysis techniques mainly focus on deriving the upper-bound number of visits to a given control location or iterations of a loop. However, there still exist gaps between such bounds and resource usage bounds. In this paper, we present a static analysis approach to derive resource usage bounds for imperative programs. We leverage techniques of program transformation, numerical value analysis, pointer analysis and program slicing, to model and analyze resource usage in a program. We have conducted experiments to derive usage bounds of various resources in C programs, including heap memory, file descriptors, sockets, user-defined resources, etc. The result suggests that our approach can infer usage bounds of resources in practical imperative programs.**

***Index Terms*—Static Analysis, Resource Bound Analysis, Resource Usage Bounds**

I. Introduction

Resource usage bounds provide a useful guidance in the design, configuration and deployment of software, especially when the software runs under a context with limited amount of resources. Unexpected or uncontrolled resource usage may degrade program performance, or even leads to CWE vulnerabilities (such as uncontrolled resource consumption, file descriptor exhaustion, etc.). Static analysis enables deriving worst-case bounds of resource usage. Most existing static bound analysis techniques 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). However, in real-world programs, resources are often manipulated via specific APIs which may involve complicated parameters (such as pointers). The usage amount of resources often depends on such parameters, and different calls to the same API may induce different usage amount of resources. E.g., when considering *free(p)* in C programs, the deallocation amount of heap memory depends on the size of the heap memory that pointer p points to.

In this paper, we present a static analysis approach to automatically derive resource usage bounds for imperative programs. We evaluate our approach by deriving usage bounds of various resources in C programs, including heap memory, file descriptors, sockets, user-defined resources, etc.

II. Approach

In this section, we describe the basic process of our approach (Fig. 1) and illustrate it via an example (Fig. 2).

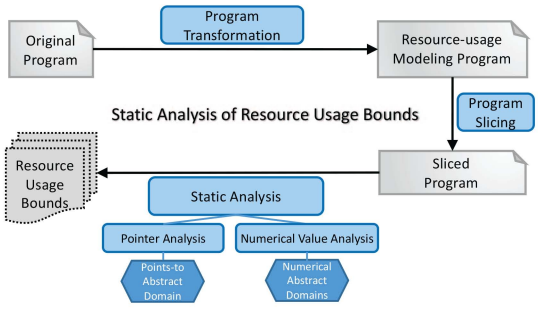

Fig. 1. Workflow of static analysis of resource usage bounds

A. Step 1: Deriving resource-usage modeling programs

First, we introduce auxiliary integer variables (called resource-usage modeling variables) to model the resource usage for resource-manipulating functions. For resource of kind $\#t$, we will introduce variables $resc\_\#t\_cur$ and $resc\_\#t\_peak$ to track respectively the current amount and the historical peak amount of $\#t$ resource usage. To model a resource-manipulating function, we will instrument auxiliary statements that update the values of resource-usage modeling variables according to the semantics of that function. E.g., to model the calling of a function allocating n number of resources of kind $\#t$, we will instrument statements like $\{\ resc\_\#t\_cur$ += n; if $(resc\_\#t\_peak < resc\_\#t\_cur)$ $resc\_\#t\_peak = resc\_\#t\_cur;\}$. To automatically instrument resource-usage modeling statements into the original program, we utilize program transformation tool Coccinelle [1]. After instrumentation, we drive from the original program the corresponding resource-usage modeling program.

B. Step 2: Resource-usage concerned program slicing

Many variables and statements in the program are useful for implementing functionality of the program but not relevant to resource usage. Based on this insight, we take strength of program slicing to reduce program size in order to improve the efficiency of subsequent analysis. We use Frama-C's Slicing plug-in [2] to conduct program slicing over the resource-usage modeling program, and keep only those program code relevant to resource usage.

2640-0715/21/$31.00 ©2021 IEEE
DOI 10.1109/APSEC53868.2021.00077

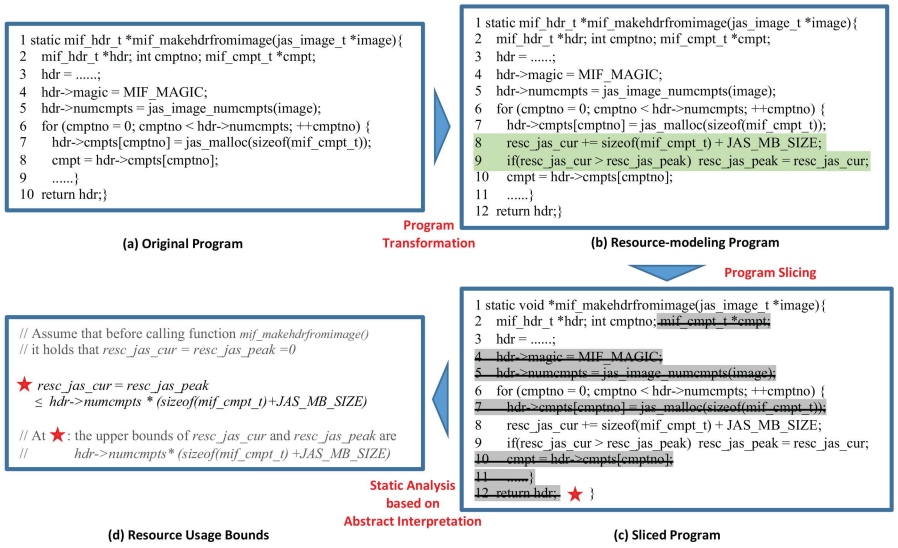

Fig. 2. Example illustration

C. Step 3: Analyzing sliced programs

We use static analysis techniques to analyze the sliced program. The main idea is to conduct resource bound analysis by combining numerical value analysis and pointer analysis under the unified framework of abstract interpretation. We use numerical value analysis based on numerical abstract domains (such as intervals, polyhedra, etc.) to infer numerical relations among resource-usage modeling variables and numerical variables in the original program. We use pointer analysis based on a points-to abstract domain to deal with resource-manipulating functions involving pointers, and help to determine the size of a resource that a pointer points to. Details of this step refer to our previous work [3].

D. Example illustration

Fig. 2 illustrates the above process via an example extracted from Jasper software. Fig. 2(a) shows the original code wherein $jas\_malloc(size)$ (in Line 7) is an allocation function (which applies to allocate $size + JAS\_MB\_SIZE$ bytes) for user-defined heap memory inside Jasper. Fig. 2(b) shows the derived resource-usage modeling program wherein the instrumented code is highlighted. Fig. 2(c) shows the sliced program wherein the deleted code is marked strikethrough. Fig. 2(d) shows the resulting resource usage bound of user-defined Jasper heap memory given by static analysis.

III. Experiments

We conduct experiments on a set of programs from open-source projects: Sod (containing many heap memory operations), FastDFS (containing many file descriptors), Redis (containing many socket operations) and Jasper (containing many user-defined resource operations). In Table I, columns "Line" and "Line′" show the number of lines of code (with all functions inlined) before and after performing slicing respectively. The column "Peak" shows the resulting upper-bounds of peak resource usages in these programs. The result shows that our approach can derive meaningful usage bounds of various resources in practical C programs, including heap memory, file descriptors, sockets, user-defined resources. The column "Time" indicates that the time overhead of our approach is also acceptable.

TABLE I
Experiment Results on Various Resources

Project	Function	Resource Type	Line	Line′	Time(s)	Peak
Fast-DFS	do_dispatch_binlog _for_threads	file descriptor	1021	428	2.89	n
	test_upload main	file descriptor	1409	165	0.83	9
Redis	anetGenericAccept	socket	139	37	0.15	1
	_anetTcpServer	socket	446	85	1.27	1
Jasper	mif_makehdr fromimage	user-defined heap	319	35	0.17	136n
Sod	BlobPrepareGrow	heap	549	173	1.09	2n+16
	make_network	heap	907	364	5.06	808n+8

IV. Conclusion

This paper presents a static analysis approach to derive resource usage bounds for imperative programs. Experimental results show its promising ability to infer the usage bounds of various types of resources in real-world programs.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Nos. 61872445, 62102432).

References

[1] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *EuroSys 2008*.
[2] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects of Computing*, pp. 1–37, 2015.
[3] G. Fan, T. Chen, B. Yin, L. Chen, T. Wang, and J. Wang, "Static bound analysis of dynamically allocated resources for c programs," in *ISSRE 2021*.