Towards Denial-of-Service Memory Vulnerabilities

Tianhan Lu[1], Yu-Ju Lee[1], Wen-Wei Liao[2,3]*

[1] Department of Computer Science, University of Colorado Boulder, CO 80309-0430, USA.
[2] University of Colorado Boulder Cooperative Institute for Research in Environmental Sciences, USA.
[3] International College of Semiconductor Technology, National Chiao Tung University, Taiwan.

* Corresponding author. Tel.: 303-497-3549; email: Wenwei.Liao@Colorado.Edu
Manuscript submitted May 4, 2019; accepted Jul. 20, 2019.
doi: 10.17706/jsw.14.9.423-436

Abstract: We address the problem of verifying a program to be free of Denial-of-Service memory vulnerabilities. More specifically, we define a program to be safe from DoS attacks if its memory usage at any time during execution is linear to sizes of its inputs. We design an analysis algorithm that verifies if a program satisfies this definition, and reports code snippets in the program that may cause a nonlinear amount of memory usage in case the verification fails. We also formally prove the correctness of our algorithm w.r.t. the above definition. Our experimental results indicate that the analysis algorithm is both effective and efficient.

Key words: Program verification, software security, static analysis.

1. Introduction

Denial-of-Service memory vulnerabilities are an important category of software security bugs, especially for server programs. Successful exploits may exhaust memory resources, leaving the software unavailable to serve requests from benign users. In this paper, we address the problem of either verifying a program to be free of Denial-of-Service vulnerabilities, or labeling which code snippets might cause DoS vulnerabilities. In particular, we concretize the definition of being free from Denial-of-Service memory vulnerabilities into that, at any time during execution, the program uses linear amount of memory usage w.r.t. sizes of its input variables.

Consider a program model that supports alias, array allocation, collection-typed variables (e.g. lists, maps, sets), mutation, and method invocation. It is challenging for static analysis techniques to produce useful results for such a program while terminate in a short amount of time. Additionally, when a verification procedure fails, it is challenging but desirable for it to return useful information, i.e. which code snippets may cause DoS vulnerabilities.

We propose an analysis algorithm that can both verify non-existence of DoS memory vulnerabilities and report suspicious code snippets back to a user if verification fails. Overall, the contributions of this paper are:

We design an analysis algorithm to verify the linear amount of memory usage of a program, and report a set of code snippets that may cause a nonlinear amount of memory usage when verification fails (Section 4).

We formally prove the analysis algorithm is sound and correct (Section 5) w.r.t. our definition of DoS memory vulnerabilities mentioned above.

The experimental results indicate that our analysis algorithm is both effective and efficient (Section 6). On average, it takes 9.5s for our tool to analyze a 21k lines-of-code program.

2. Motivating Examples

In this section, we present the results from our analysis algrithm over a few example programs that motivated our analysis algorithm. Consider Example 1 in Fig1. Our analysis will report verification failure on this program, and report the loop at line 25 as a code snippet that potentially increase memory usage for a nonlinear amount. Intuitively, we consider this program as vulnerable to DoS attacks because method argument T is controlled by a user and hence the amount of memory usage is controlled by a user.

Consider Example 2 in Fig. 2. Our analysis will report verification success on this program. Intuitively, the while loop at line 3-6 will be iterated for the length of list l2. Note that if line 5 is replaced by cons([1], [2]), then our analysis will report verification failure, because the amount of memory increment caused by the loop will be nonlinear.

```
1 Unit method(List[Int] l, int x) {
2     while (x > 0)
3         cons(1, x);
4         x = x - 1;
5 }
```

Fig. 1. Example 1.

```
1 Unit method(List[Int] l1, List[Int] l2) {
2     iter = iter(l1);
3     while (hasnxt(iter)) {
4         x = next(iter);
5         cons(l2, x); // cons(l1, l2);
6     }
7 }
```

Fig. 2. Example 2

Consider Example 3 in Fig3. Our analysis will report verification success on this program. Intuitively, although method fibinacci is recursive and the maximum number of times executing the method is not trivial to obtain, this method does not increase memory usage.

Consider Example 4 in FFig. 4. Our analysis will report verification failure on this program. Intuitively, the number of times executing loop at line 6-13 is difficult to obtain (in fact it is exponential to variable number) and the loop allocates an array at line 9. Therefore, the amount of memory increase is not linear to input values.

3. Definition

3.1. Language

In this subsection, we formalize the core language (see Fig. 3). The supported types are Integer type Int, Boolean type Bool, Unit type Unit, Array type Arr[τ], List type List[τ], and Iterator type Iter[τ]. In particular, the list type is used to model collection types such as sets, lists and maps. A program L is defined as a main method I à Ô Ü à together

with a list of methods I%. A method I consists of its return type ì, its name / , its list of arguments T§ and its body statement O. As usual, a statement can be assignment T L A, method invocation T L I : T§, array allocation T L newarr ì >A?, array read T L Tñ>A?, array write T>A? L A, sequential composition of statements Q3 â Q6, disjunction

if : A then Q5 else Q6, loop while : A; Q, return statement ret T and skip statement skip. Moreover, we have statements

modeling collection operations. Statement T L iter : Tñ; creates an iterator value (which is associated with list Tñ) and assigns it to variable T. Statement T L next : Tñ; takes the next element from iterator Tñ and assign it to variable T. Statement cons: TáTñ; appends list- or non-list-typed variable Tñ to list-typed

variable T, thus modeling list concatenation as well as list append. Expressions can be variables, values, arithmetic and comparison operations, negation, and querying if an iterator has reached the end, i.e. hasnxt : T;. Values include integers, booleans, addresses, iterator values iter : Já=; (where n denotes the current index of iterator and a denotes the address of the corresponding list value), array values >Rá® áR? and list values ÃRá® áRÄ.

3.2. Operational Semantics

In this subsection, we formalize the small-step operational semantics (see Fig. 6) of the language defined in Section 3.1. The operational semantics defined here formally describes how an abstract machine executes a program written in the language defined in Section 3.1. Additionally, they will be used in Section 5 to prove that the analysis defined in Section 4 is correct, i.e. Theorem 2. Intuitively, the operational semantics defined in Fig 6 respects the semantics of a Java or C++ program, but without the Object-Oriented feature. Scalar values (i.e. integer, booleans, addresses and iterators) reside in the stack and other values (i.e. array and list) reside in the heap.

```
1  Unit fibonacci(int number) {
2       if(number == 1 || number == 2)
3            return 1;
4       return fibonacci(number-1) + fibonacci(number -2); // recursion
5  }
```

Fig. 3. Example 3.

```
1  Unit fibonacci(int number, List[Int] l) {
2       if (number == 1 || number == 2)
3            return 1;
4       fibo1=1; fibo2=1; fibo=1;
5       i= 3;
6       while (i<= number){
7            //Fibonacci number is sum of previous two Fibonacci number
8            fibo = fibo1 + fibo2;
9            arr = newarr Int[1];
10           arr[0] = fibo;
11           cons(l, arr);
12           fibo1 = fibo2;
13           fibo2 = fibo;
14           i = i + 1;
15      }
16 }
```

Fig. 4. Example 4

A program state $state$ is defined as pair of stack frames and a heap â§é?Notation â§ denotes a list of stack frames Ãâ$_5$á® áâ$_á$Ä where each stack frame is a mapping from variables to values, i.e. â >T p R?. Heap é is a mapping from addresses to values é >= p R?. We sometimes denote list Ãâ$_5$á® áâ$_á$Ä as â^Û ø â^ç where â^Û denotes the head of list (i.e. â$_5$) and â^ç denotes the tail (i.e. â$_6$á® áâ$_á$). A program's initial state ê$_4$ is defined as >â$_4$áé$_4$? Initial stack â$_4$ is defined as >T^4 p R^4? where T^4 denotes input variables and R^4 denotes input values. Initial store é$_4$ is defined as >=^{4} p R^4? where =^{4} denotes addresses that point to input values R^4. Note that we assume the program is already type-checked before being executed. As a result, in the rules below (see Fig 6) each variable is typed and their types are available during execution.

To simplify the representation of rules in Fig 6, we define a notation ê>T p R? to denote value mutation that happens at the top frame â^Û of a program state ê : â^Û ø â^ç áé;, i.e. ê>T p R? : â^Û>T p R? ø â^ç áé;. Notation â>T p R? denotes a stack that is the same as stack â, except that variable T is mapped to value

R. We also define a similar notation é>= p R?to denote a heap that is the same as heap é, except that address = is mapped to value R. We use notation T ÷ ì to denote that variable T's type is τ.

Rule E-Method creates a new stack frame before invoking the method by copying values from the current stack frame. The return value is the only value that is stored in the top stack frame after method invocation, which is guaranteed by Rule Ret. Rule E-NewArray allocates a new chunk of space in the store. Rule E-ArrayW writes a value into an array value in the store. Rule E-ArrayR reads a value from an array value in the store. Rule E-Iter creates an iterator value that is associated with a list. Rule E-Next retrieves a value from a list value in the store. Rule E-ConsAppend appends a value into a list value in the store, based on the types of two arguments. Rule E-ConsConcat concatenates a list value into another list value in the store, based on the types of two arguments. The remaining rules are standard and hence not further explained. The small-step operational semantic rules for evaluating expressions are standard and hence omitted.

3.3. Size Semantics

In this subsection, we define the memory usage R of a value w.r.t. a heap é, i.e. the *reachable size* of value R in the heap é, as well as the memory usage ê of a program state (see Fig. 7). Intuitively, the size of a list or array value is the size of the reachable heap starting from the value. Similarly, the size of a state is the size of the reachable heap starting from list and array values stored in the stack. Note that we do not take into consideration the sizes of scalar values (i.e. integers, booleans, and iterator values), because our observation is that memory usage can only be significantly increased by list values and array values. Notation Dom : â; denotes the domain of mapping â and Dom : é; denotes the domain of mapping é.

3.4. Problem Definition

Formally, the problem that we address in this paper is defined as follows. Given a program L and an initial state ê$_4$ L >â$_4$ áé$_4$?(which instantiated with input values R$_5^4$ á® áR$_á^4$), we either verify that the program's memory usage at any

point of execution is linear to the total size of input values (see Fig. 8), or if the verification fails, output a set of statements that may cause the constraint to be violated. Note that initial state ê$_4$ trivially satisfies the constraint in Fig. 8 by setting all M$_Ü$ to 1.

Program	p	::=	m_{main}; $\overline{m}$
Method	m	::=	$\tau\ M(\bar{x})\{s\}$
Statements	s	::=	$x = e \mid x = M(\bar{x}) \mid x =$ newarr $\tau[e] \mid x = x'[e] \mid x[e] = e'$
			$\mid x =$ iter $(x') \mid x =$ next $(x') \mid$ cons (x, x')
			$\mid s_1; s_2 \mid$ if(e)then s_1 else $s_2 \mid$ while(e) $s \mid$ ret $x \mid$ skip
Expression	e	::=	$x \mid v \mid e_1 \oplus e_2 \mid e_1 \bowtie e_2 \mid e_1 \vee e_2 \mid \neg e \mid$ hasnxt (x)
Value	v	::=	$n \in \mathbb{Z} \mid b \in \{$true, false$\} \mid a \in Addr$
			$\mid$ iter $(n^{\in \mathbb{N}}, a^{\in Addr}) \mid [v, \cdots, v] \mid \langle v, \cdots, v \rangle$

(a) The core calculus. Operator $\oplus$ and $\bowtie$ stand for arithmetic and comparison operations. M denotes a string.

Types	τ	::=	Int $\mid$ Bool $\mid$ Unit $\mid$ Arr$[\tau]$ $\mid$ List$[\tau]$ $\mid$ Iter$[\tau]$

(b) Types.

Fig. 5. Language definition.

4. Algorithm

This section presents the analysis algorithm in the form of rules (see Fig. 10). The analysis takes as input a program and apply Rule T-Program, which will apply other rules. Specifically, judgement form _ O÷ : á#; means that the algorithm takes as input statement O, and then outputs a set of statements , such that executing any of them may cause a program state not to preserve its memory usage as a linear combination

of input values, as well as the type of memory usage # of statement O. Note that for all rules in Fig 10, it is assumed that before applying a rule, the precondition is that we have Theorem 1 hold true. In other words, each rule infers the output and # only if Theorem 1 holds before applying the rule. Also note that the analysis algorithm is parametrized by two procedures ConstProp and LoopBound. Procedure ConstProp takes as input a program and outputs the set of expressions that must be constant for any input values. Procedure LoopBound takes as input a loop and outputs the symbolic loop bound (i.e. the maximum number of times executing the loop body).

We define the type of memory usage # of a statement • in Fig. 9a. If a statement • is of type ÆÇâÂáÖ then if starting from any program state, executing the statement will not increase the size of the program state. If a statement O is of type ÆÅáÖ¼âáæç then if starting from any program state, executing the statement will increase the size of the program state by a constant. Note that we define type ÆÇâÂáÖ as a subtype of type ÆÅá¼Ö¼âáæç in Fig. 9b, because when M is 0, a statement satisfying the definition of type ÆÅáÖ¼âáæç also satisfies the definition of type ÆÇâÂáÖ. If a statement O is of type ÆÅáÖ¼âáæç then if starting from any program state, executing the statement will increase the size of the program state by a linear combination of input values, together with a constant. Note that we define type ÆÅáÖ¼âáæç as a subtype of type ÆÅáÖÅÙá in Fig. 9b, because when all M_j are 0, a statement satisfying the definition of type #MlaPgl also satisfies the definition of type ÆÅáÖ¼âáæç. If a statement does not satisfy any of the above three

definitions about the type of memory usage, we define it to be of type #MlaEp`grp_pW denoting that executing the statement may arbitrarily increase the memory usage of a program.

Next we explain the rules. Notation T denotes the size of a value in a concrete heap, where the value is mapped from variable T in a concrete stack. We use notation Dom L; to denote the set of variables defined in all methods of program L.

Rule T-Next, T-Iter, T-Assign, T-ArrayR, T-Ret trivially do not increase memory usage of a program state. Rule T-ConsAppend can at most increase memory usage by a linear amount, because it is assumed that variable T" will use a linear amount of memory (recall that it is assumed Theorem 1 holds true before applying each rule). Similarly, Rule T-ConsConcat can at most increase memory usage by a linear amount. Rule T-NewArrayConst applies when procedure ConstProp infers that expression A is always a constant for any input values. As a result, memory increase is a constant. Rule T-NewArrayConst applies when procedure ConstProp cannot guarantee that expression A is always a constant for any input values. As a result, we conservatively assert that the amount of memory increase is arbitrary. Rule T-ArrayW increases memory usage by a linear amount, because 1) if expression A" is a scalar value, then sizes of scalar values are constant (see Fig. 7), 2). if expression A" is a non-scalar value, then according to the assumption Theorem 1, the memory increase will be a linear amount.

Rule T-WhileLin1 applies when procedure LoopBound infers that the loop's bound is a linear expression using any variable defined in the program and when the memory usage type of the loop body is a subtype of type ÆÅáÖ¼âáæç i.e.

the amount of memory increment caused by executing the loop body once is at most a constant. Rule T-WhileLin2 applies when procedure LoopBound infers that the loop's bound is a linear expression using any variable defined in the program and when the memory usage type of the loop body is a supertype of type ÆÅáÖÅÙá. Rule T-WhileNonLin1 applies

when procedure LoopBound cannot infer that the loop's bound is a linear expression using any variable defined in the program and when the memory usage type of the loop body is a subtype of type #RmMla. Rule T-WhileNonLin2 applies

when procedure LoopBound cannot infer that the loop's bound is a linear expression using any variable defined in the program and when the memory usage type of the loop body is a supertype of type #MlaGmlqr

Rule T-Method applies when the invoked method / is not a recursive function. Intuitively, assigning the return value to variable can at most increase memory usage (when compared with the memory usage before invocation) by a same amount as executing method. Rule T-MethodRec applies when the invoked method / is a (mutually) recursive function. Intuitively, we first collect the set of all possible invocation relations starting from method /, i.e. which method may invoke which method. Then we define a constraint that enforces that, if method / invokes / then

- The set of statements (that possibly increase memory usage nonlinearly) inferred from method / must be included in the set inferred from method / because if a statement from method / may cause
- vulnerabilities then method / should also report it (because method / may invoke method /). The memory usage type of method / should be at most as restrictive as the type of method / (because method / may invoke method /).

E-WhileT E-WhileF E-Assign

E-Method

E-Ret E-SeqSkip E-Seq

E-IfT E-IfF

E-NewArray E-ArrayW

E-ArrayR E-Iter

E-Next E-ConsAppend

E-ConsConcat E-HasNext

Fig. 6. Small-step operational semantics.

In the end, we try to obtain a least fix point solution to the above constraint and use the solution as the output of this rule. A least fix point preserves analysis precision as much as possible.

Rule T-Seq and T-If first infer types of memory usage and sets of statements (that possibly increase memory usage nonlinearly) for each sub-statement, and then join the types (i.e. choose the least restrictive type to guarantee soundness) and union the sets.

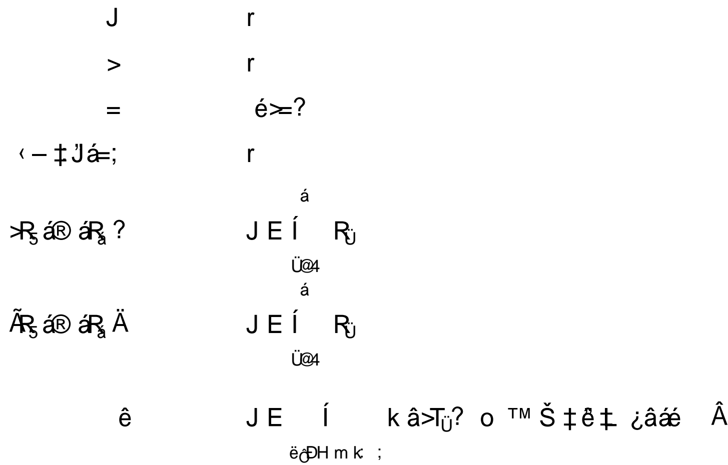


Fig. 7. Size semantics

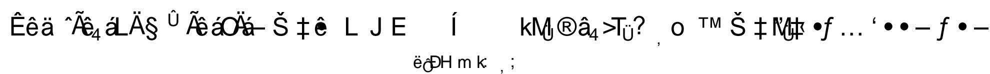
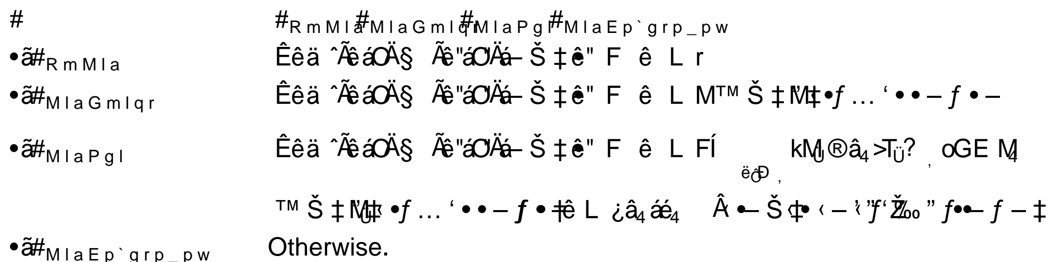


Fig. 8. (Problem definition) Linear memory usage



(a) Types.

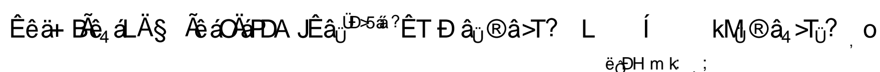
(b) Type lattice.

Fig. 9. Types of memory usage of a statement

5. Soundness

In this section, we present the soundness theorems that establish the correctness of the analysis algorithm defined in Section 4 w.r.t. the problem definition in Section 3.4 and the operational semantic defined in Fig. 6, and formally prove them.

Theorem 1 (Linearity). *Given program* L, *if* _ Lã Î á#; *and the program starts with initial state* ê4 *, then*



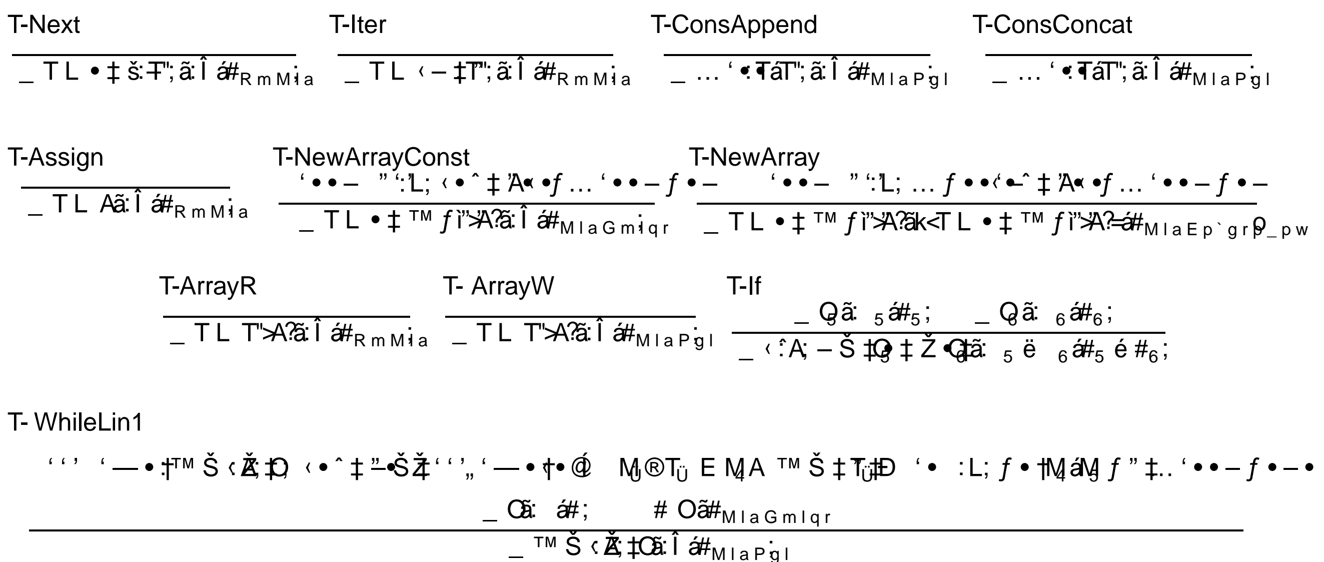
Proof. We prove this theorem by a mathematical induction on each rule defined in the algorithm (see Figure 10).

Base case. It is trivial that the above theorem holds for any initial state, because the left hand side and right hand side are syntactically the same.

Inductive cases. Next we prove the above theorem holds true for any state that is reachable from the initial state.

The inductive assumption is that the above theorem is valid before applying each rule.

- Rule WhileLin1: Since the memory usage type of the loop body is a subtype of type, the amount of memory increment for all variables is at most a constant. Since the loop bound is a linear expression using sizes of program variables (which are linear expressions of input values according to the inductive assumption) and the total amount of memory increment of each variable is a product of the above two, the total memory usage increment for each variable is at most a linear amount.
- Rule WhileLin2: Since the memory usage type of the loop body is a supertype of type, the amount of memory increment for all variables can be at least a linear expression of variable sizes. Since the loop bound is a linear expression using sizes of program variables (which are linear expressions of input values according to the inductive assumption) and the total amount of memory increment of each variable is a product of the above two, the total memory usage increment for each variable can be arbitrary.
- Rule WhileNonLin1: Since the memory usage type of the loop body is a subtype of type, the amount of memory increment for all variables is 0. As a result, the total amount of memory increment for all variables is 0.
- Rule WhileNonLin2: Since the loop bound can be arbitrary, the total amount of memory increment for all variables can also be arbitrary.
- Rule T-Method: Any memory increment caused by invoking method may escape to the caller, which is exhibited by outputting the memory usage type of statement.



T- WhileLin2

[illegible]

T- WhileNonLin1

[illegible]

T- WhileNonLin2

[illegible]

T-MethodRec

[illegible]

T-Method

[illegible]

T-Ret

[illegible]

T-Seq

[illegible]

T-Program

[illegible]

Fig. 10.Analysis algorith.

- Rule T-MethodRec Constraint % guarantees that if method [illegible] invokes [illegible], then the amount of memory increment for statement [illegible] is at least as much as for statement [illegible], thus considering the possibility that memory increment during invoking method [illegible] might escape to its caller method [illegible]. The least fix point solutionpreserves as much precision as possible.
- Rule T-Next, T-Iter, T-Ret, T-Assign, T-ArrayR: According to size semantics defined in Figure 7, there is no memory increment.
- Rule T-ConsAppend According to the inductive assumption, the amount of memory usage of variable T" is at most a linear expression of input values. Hence the amount of memory usage variable T shall remain a linear expression of input values.
- Rule T-ConsConcat According to the inductive assumption, the amount of memory usage of variable T" is at most a linear expression of input values. Hence the amount of memory usage of variable T shall remain a linear expression of input values.
- Rule T-NewArrayConst Since procedure ConstProp infers that expression A is always a constant for any input values, the amount of memory increment is a constant.
- Rule T-NewArray: Since procedure ConstProp cannot infer that expression A is always a constant for any input values, we conservatively assume that the amount of memory increment can be

arbitrary.

- Rule T-ArrayW: If expression A" is scalar-typed (i.e. integer, boolean, and iterator), then the amount of memory increase is 0.
  Otherwise if expression A" is not scalar-typed (i.e. list, array), then according to the inductive assumption, the amount of memory usage of variable A" is at most a linear expression of input values. Hence the amount of memory usage of variable T shall remain a linear expression of input values, soundly disregarding the memory decrement by removing the pointer to value $R_Ü$ (see Rule E-ArrayW)
- Rule T-Seq The join of memory usage types $\#_5$ and $\#_6$ returns the least restrictive type, i.e. a more conservative memory usage type among types $\#_5$ and $\#_6$ that over-approximates the amount of memory
  increment.
- Rule T-If: The join of memory usage types $\#_5$ and $\#_6$ returns the least restrictive type, i.e. a more conservative memory usage type among types $\#_5$ and $\#_6$ that over-approximates the amount of memory increment.

Theorem 2 (Soundness). *Given program* L, *if* _ Lã: Î á#; *and the program starts with initial state* $ê_4$, *then*

$$Êêä+ Ãêå_4 áLÄ§ Ãêá ÓÄáFDA J ê L í kM_Ü®â_4 >T_Ü? o SDA NM_Ü E ⊖? K J O P = J P \quad ë_ÚÐH m k ;$$

Proof. The proof can be directly obtained after proving Theorem 1 and using the definition of ê in Fig. 7.

6. Experiment

6.1. Implementation

We implemented the algorithm defined in Fig. 10 using T.J. Watson Libraries for Analysis (WALA)[1] targeting at Java programs. Since the Object-Oriented feature of Java is not modeled in the core calculus defined in Fig. 6 and hence not considered in the algorithm, the experimental results should be interpreted as sound and correct without considering the Object-Oriented feature.

In the implementation, procedure ConstProp adopts a simple strategy, i.e. if an expression is itself a constant, then return true and otherwise return false. Procedure LoopBound tries to syntactically identify loops that iterate over a collection- or array-typed variable. The solver used in Rule T-MethodRec makes a most conservative estimation, i.e. as long as a method may be recursive and may increase memory usage, consider it as being able to increase memory usage arbitrarily.

6.2. Experimental Results

The set of Java programs come from Space and Time Analysis for Cybersecurity (STAC) [3]. The programs contain memory exhaustion vulnerabilities and others do not. The experiments were run on a MacBook ' " ' ï s w á t. The machine has a 4-core 6th generation Intel i7 CPU, 16GB memory and 1TB SSD. Running time was measured by the Real time (wall clock time) in the output of time command.

To evaluate our technique, we propose the following two research questions:

Effectiveness. How effective is our technique in verifying linear amount of memory usage or reporting code snippets that may cause Denial-of-Service vulnerabilities? As shown in the second column Figure 11, our technique is able to verify that only 11.41% of all loops in the benchmarks might cause nonlinear amount of memory usage. Moreover, as shown in the third column, only 0.72% of all methods are recursive methods and might cause nonlinear amount of memory usage. As a result, only these reported loops and

[1] http://wala.sourceforge.net/wiki/index.php/Main _Page

methods need a secure analysis to manually look into, thus excluding the vast majority of the code.

Efficiency. How fast is our verification algorithm? As shown in Fig 12, on average it took 9.5 seconds for our tool to analyze a 21k lines-of-code program, which we consider as indicating that our algorithm can scale to real life programs.

Benchmark name	Number of vulnerabilities	Vulnerable loops	Recursive methods
Airplan 5	1	35/299 (11.17%)	2/1848 (0.11%)
InfoTrader	0	11/133 (8.27%)	12/1041 (1.15%)
Linear Algebra	1	14/81 (17.28%)	0/221 (0.00%)
Malware Analyzer	0	2/15 (13.33%)	4/79 (5.06%)
Powerbroker 1	0	8/129 (6.20%)	15/1532 (0.97%)
Powerbroker 4	0	8/115 (6.96%)	14/1493 (0.94%)
Tweeter	1	16/124 (12.90%)	3/566 (0.53%)
Withmi 3	0	14/96 (14.58%)	8/1294 (0.62%)
Withmi 4	0	16/95 (16.84%)	9/1258 (0.71%)
Average		(11.17%)	(0.11%)

Fig. 11.Effectiveness: Suspicious code snippets.‘Ž—•• ò —Ž•‡”ƒ„Ž‡ Ž‘‘’•ó ’”‡•‡•– –Š‡ ’‡” in all loops that may cause nonlinear amount of memory increment. ‘Ž—•• ò ‡…—”•‹˜‡ •‡–Š‘†•ó • percentage of recursive methods in all methods that allocates memory, which are all considered to may increase the memory usage with a nonlinear amount (see Section 6.1} for detail).

Benchmark name	Lines of code	Time consumption
Airplan 5	25869 loc	11.165s
InfoTrader	29335 loc	8.419s
Linear Algebra	3817 loc	8.236s
Malware Analyzer	2605 loc	5.943s
Powerbroker 1	31705 loc	10.696
Powerbroker 4	31625 loc	10.053
Tweeter	9896 loc	8.128s
Withmi 3	26072 loc	11.632s
Withmi 4	25972 loc	11.227s
Average	20766.22 loc	9.5s

Fig. 12.Efficiency: Running time.

6.3. Discussion and Limitations

Since the analysis algorithm defined in Fig 10 is flow- and context-insensitive, its precision can be improved by inferring or verifying pre-conditions before applying each rule. Additionally, Rule T-Method is conservative under the program model of passing references (instead of values) back to a caller, i.e. a nonlinear amount of memory allocated in a callee method does not necessarily escape back to the caller

method.

7. Related Works

The most relevant works are in the area of resource and bound analysis using static program analysis techniques. While these works do not return useful information about resource usage if the analysis fails, our algorithm is designed with this feature. Additionally, some of these works may complement our work in the sense that, the procedure LoopBound defined in Figure 10 may be implemented with these works. Below are more detailed arguments supporting the above claims.

Sized-types. The works of Pedro B Vasconcelos *et al.* [15], [16] take a type-based approach by traversing program statements and generating constraints in a particular form w.r.t. variable sizes in a syntax-directed way, and finally obtaining closed-form solutions to the constraints using an external solver. The solution to the constraints is the symbolic resource usage of the given program. Compared with our work, these works define a functional core calculus that behaves differently than ours from the perspective of memory usage, i.e. while the fact that memory usage may be affected by mutation is not modeled by a functional core calculus, it is modeled by our core calculus. Since an analysis designed for such a functional core calculus [15], [16] does not take this effect into consideration, these works do not work on the core calculus defined in this paper (see Fig. 5 and 6), thus not giving useful information to secure analysts.

Cost analysis for Java programs. The works of Elvira Albert *et al.* [1]-[3] take a similar approach as Pedro B Vasconcelos *et al.* [15], [16] by generating constraints w.r.t. variable sizes in a syntax-directed way and finally obtaining closed-form solutions to the constraints using an external solver. The closed-form solution to the constraints is the symbolic resource usage of the given program. The difference here is that apart from generating constraints w.r.t. variable sizes, this line of works also path-insensitively generate constraints w.r.t variable values, thus achieving more precision. The work of Diego Esteban Alonso-Blas et al. [4] extends the work of Elvira Albert *et al.* [1]-[3] by defining a more expressive form of constraints w.r.t. variable sizes. The work of Antonio Flores-Montoya *et al.* [8] extends the work of Elvira Albert *et al.* [1]-[3] by excluding invalid constraints w.r.t. variable sizes (via identifying infeasible control flow paths) and hence improves analysis precision. Compared with our work, this line of works are more precise in terms of inferring resource bounds, because they collect constraints w.r.t. both variable values and sizes. However, recall that the problem addressed by this paper (see Section 1) is to either verify a program to use a linear amount of memory usage or report its vulnerable code snippets. For this purpose, this approach is not suitable because it is neither guaranteed that an external solver can return a closed-form solution (i.e. symbolic resource usage bounds), nor the solver can ever terminate.

When any of the two situations happens, this analysis will not give any helpful information to a secure analyst.

RAML. The works of Jan Hoffmann *et al.* [12], [13] targets at functional programs. It defines polynomial bound templates with unknown coefficients, and then uses a type-directed approach to collect constraints on the coefficients from a particular form of the language construct (i.e. a recursive call in pattern matching). Eventually, the set of collected constraints become a linear programming problem and is solved by an external solver. On one hand, this approach is similar to our work because both approaches take advantage of the insight that, traversing collection-typed variables is a very common coding pattern and can be used to infer resource usage of a program. On the other hand, similar to the work of Pedro B Vasconcelos et al. [15], [16], since this technique works for a functional core calculus, it does not consider mutation's effects on memory usage, thus not returning useful information to secure analysts.

SPEED. The works of Sumit Gulwani *et al.* [9]-[11] provide various ways to abstract program transitions such that the time bound can be directly inferred from the resultant transitions. Meanwhile, the abstraction

of transitions is generated by and is guaranteed to be sound (i.e. semantic-preserving) via Abstract Interpretation [7] based techniques. The work of Florian Zuleger [17] extends the works of Sumit Gulwani *et al.* [9]-[11] by adopting a new numeric domain for abstract interpretation, i.e. size-change abstraction. The domain is less expensive than other numerical domains, and yet at most times it is sufficient to effectively infer time bounds. This lines of work are complement to our work, because they can be used as an analysis module (i.e. procedure LoopBound) in the algorithm (see Fig 10) to infer linear loop bounds.

Alternate runtime and size analysis. The work of Marc Brockschmidt [5] alternatively infers and refines variable sizes and timebounds for each Strong-Connected-Component in a program. This approach is complement to our work because procedure LoopBound defined in Fig. 10 may be implemented with this technique.

Compositional analysis. Similar to the work of Jan Hoffmann *et al.* [12], [13], the work of Quentin Carbonneaux [6] first defines bound templates with unknown coefficients for a given program, and then collect constraints on the coefficients. The set of constraints finally become a linear programming problem and are solved by an external solver. The difference between this work and the work of Jan Hoffmann et al. [12], [13] is that, this work targets at imperative programs while the other targets at functional programs. This work is not suitable for the purpose of verifying a program to use a linear amount of memory usage or reporting its vulnerable code snippets, because if a program's resource bound does not fit the bound templates, then this analysis cannot provide any useful information to a secure analyst (that is concerned with DoS memory vulnerabilities of the given program).

Simple bound analysis. The work of Moritz Sinn [14] takes advantage of ranking functions to infer time bound of a program. Ranking functions and transition abstraction [9]-[11], [17] are similar in the sense that, they both semantically abstract a program transition into a simpler form, from which a time bound is easier to infer. As a result, this work is also complement to our work because it can be used as a module (i.e. procedure LoopBound) in the algorithm (see Fig 10) to infer linear loop bounds.

8. Conclusion

In this paper, we address the problem of Denial-of-Service memory vulnerabilities from one perspective, i.e. a program may only exhibit a memory usage linear to its input values. We present an analysis algorithm to verify a program to satisfy this definition, or report code snippets that may cause the program to use a nonlinear amount of memory if the verification fails. We then formally prove the correctness of the analysis algorithm. Our experimental results also indicate that our analysis algorithm is bot effective in verification and efficient in analysis time.

References

[1] Elvira, A., Puri, A., Samir, G., Germán, P., & Damiano, Z. (2007). Cost analysis of java bytecode. *Proceedings of the European Symposium on Programming*.

[2] Elvira, A., Samir, G., & Miguel, G. Z. (2007). Heap space analysis for java bytecode. *Proceedings of the 6th international symposium on Memory management*.

[3] Elvira, A., Samir, G., & Miguel, G. Z. (2009). Live heap space analysis for languages with garbage collection. *Proceedings of the 2009 International Symposium on Memory Management*.

[4] Diego, E. A. B., & Samir, G. (2012). On the limits of the classical approach to cost analysis. *International Static Analysis Symposium*.

[5] Marc, B., Fabian, E., Stephan, F., Carsten, F., & Jürgen, G. (2014). Alternating runtime and size complexity analysis of integer programs. *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[6] Quentin, C., Jan, H., & Zhong, S. (2015). Compositional certified resource bounds. *ACM SIGPLAN Notices, 50(6)*, 467 478, 2015.

[7] Patrick, C., & Radhia, C. (1997). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM Sigact-sigplan Symposium on Principles of Programming Languages*.

[8] Antonio, F. M., & Reiner, H. (2014). Resource analysis of complex programs with cost equations. *Asian Symposium on Programming Languages and Systems*.

[9] Sumit, G., Sagar, J., & Eric, K. (2009). Control-flow refinement and progress invariants for bound analysis. *ACM Sigplan* Notices, *44(6)*, 375 385.

[10] Sumit, G., Krishna, K. M., & Trishul, C. (2009). Speed: Precise and efficient static estimation of program computational complexity. *ACM Sigplan Notice*s.

[11] Sumit, G., & Florian, Z. (2010). The reachability-bound problem. *ACM Sigplan Notice*s.

[12] Jan, H., Klaus, A., & Martin, H. (2011). Multivariate amortized resource analysis. *ACM SIGPLAN Notices*.

[13] Jan, H., Ankush, D., & Shu, C. W. (2017). Towards automatic resource bound analysis for ocaml. *ACM Sigplan Notices*.

[14] Moritz, S., Florian, Z., & Helmut, V. (2014). A simple and scalable static analysis for bound analysis and amortized complexity analysis. *International Conference on Computer Aided Verification*.

[15] Pedro, B. V. (2008). *Space Cost Analysis Using Sized Types*. PhD thesis, University of St Andrews.

[16] Pedro, B. V., & Kevin, H. (2003). Inferring cost equations for recursive, polymorphic and higher-order functional programs. *Proceedings of the Symposium on Implementation and Application of Functional Languages*.

[17] Florian, Z., Sumit, G., Moritz, S., & Helmut, V. (2011). Bound analysis of imperative programs with the size-change abstraction. *International Static Analysis Symposium*.