An Approach for Identifying Detecting Objects of Null Dereference

Yukun Dong
College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China
Email: dongyk@upc.edu.cn

***Abstract*—On account of the complexity of programs, it is difficult to identify all detecting objects of null dereference, which is one of the preconditions of null dereference detection. This paper introduces an approach for identifying all detecting objects of null dereference of C programs. First, based on the relationship of dereference expressions with nodes of abstract syntax tree (AST), we identify referenced pointers; then based on the abstract storage described by region-based three value logic (RSTVL) and function summary, we identify detecting objects of null dereference. In order to validate the adequacy of our approach, five real-world projects are utilized for experimental analysis, and the results show that our approach could identify all detecting objects of null dereference.**

***Index Terms*—null pointer dereference, defect detection, addressable expression, function summary**

I. INTRODUCTION

With increasing of software scale and complexity, software security becomes increasingly apparent. Particularly, null dereference has become one of the main causes of software security vulnerabilities, and it is one of the most common and difficult defects to eliminate.

At present, null pointer testing methods can be divided into dynamic methods [1, 2] and static methods [3-7]. Static methods check pointers dereference on the precondition of without running programs, which can be divided two categories: null dereference detection [3-5] and dereference validation [6, 7]. Generally, null dereference detection will first implement dataflow analysis or points-to analysis, then check if the pointer being referenced is null based on the analysis result; dereference validation is demand-driven, identify the pointer being referenced first, then analyses along the control flow backwards from the program point of a pointer dereference, checks if the pointer being referenced may be null.

Both static null pointer testing methods need to identify pointers being referenced, and identify detecting objects of null dereference based on associations between expressions. It is difficult to identify all detecting objects of null dereference, because pointer, struct and array exist in C programs, which cause alias, hierarchical, logic relationships exist among variables, and pointer parameter, especially complex type parameter.

If some detecting objects of null dereference unidentified, will lead to false positive of null dereference defects. The difficulties of identification lie in two aspects: First, some pointer expressions have complex grammatical structure; second, complex relationships among expressions, including alias, hierarchy, parameters with arguments, etc.

To solve these problems, we first establish mapping relationship between addressable expressions [8] with nodes of AST, and then apply RSTVL [9] to describe memory state of any memory object and all kinds of associations. Based on the analysis result, we identify detecting objects of null dereference by the following two steps. At the first step, we identify pointer expressions from AST based on the mapping between addressable expressions and nodes of AST, so we identify referenced pointers; at the second step, we identify detecting objects of null dereference for each pointer being referenced based on the result of data flow analysis and function summary [10].

This paper makes the following contributions:

- We introduce an approach for identifying pointer expressions from AST based on the relationship of addressable and nodes of AST.
- We show how to identify various detecting objects of null dereference based on RSTVL and function summary.

The remainder of this paper is organized as follows. Section II presents background on defect detection and motivation examples. Section III introduces addressable expression and RSTVL. Section IV and Section V introduce identifying detecting objects of null dereference. We present experimental results in Section VI, related work in Section VII and conclusion is in Section VIII.

II. BACKGROUND AND MOTIVATION

During a program execution, the temporal safety property indicates a series of operations that must be executed in a specified manner.

Definition 1: *Defect pattern*. Syntax or semantics feature presented by defect that occurred frequently in programs.

Defect pattern describe a kinds of property of program, satisfy it will lead defects. For example, null dereference as a defect pattern appears to be a null pointer is referenced.

Definition 2: *Defect feature*. For a defect pattern, it can detect whether some properties violate syntax or

doi:10.4304/jsw.9.11.2771-2777

semantics rules of program, these properties related variables are defect features of defect pattern.

Defect feature can be understood whether defect detection related to some addressable expressions, and these addressable expressions are called detecting objects.

Definition 3: *Detecting object of null dereference*. A pointer with definite points-to attribute and related to null dereference defect detection.

We have implemented a defect detecting tool DTSGCC, which is a defect testing system for C written in Java. DTSGCC analyzes programs in five stages as shows in Figure 1. The last step of analysis stage is defect detecting, all detecting objects of null dereference can be identified in this step.

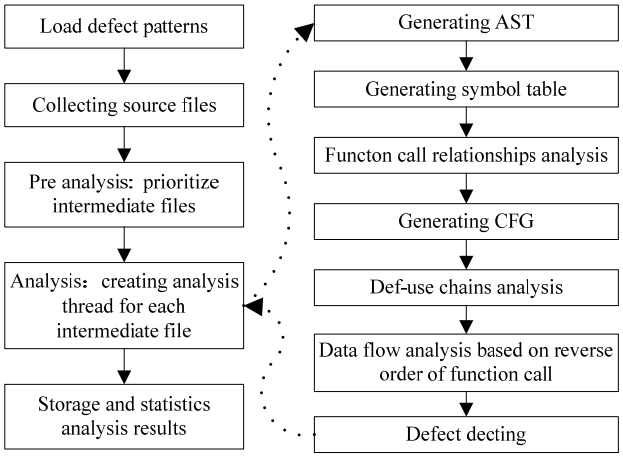

Figure 1. Analysis stages of DTSGCC

We use two examples in Figure 2 to illustrate some obstacles of identifying defecting objects of dereference. For the example of Figure 2(a), pointer expression **pst*[*i*]->*m* at line 7 actually implies three pointers being referenced: *pst*, *pst*[*i*], *pst*[*i*]->*m*, all of which need to be identified, or may lead to false negative null dereference defect. But *pst*[*i*] and *pst*[*i*]->*m* are not top-level variable, they can not be identified by analysing variable declaration, it's not easy to identify them.

For the example of Figure 2(b), *p* is referenced at line 4, there is an alias between *p* and *ps->a*, since *ps* is a formal parameter, we can't deduce the real point information of *ps->a* in function *f2* and wheather *ps->a* is null pointer. It can only be determined based on the calling context at call site. Because *ps->a* is not top-level parameter, the mapping between of parameters with arguments is unknown because the hierarchy and alias relationship between expressions. In fact, *f3* calls *f2* at line 9, and *s.a* maps *ps->a*, if we check whether *p* is referenced safely at line 4, we should check whether *s.a* is a safe pointer at line 9, and treat *s.a* as a detecting object of null dereference.

```
typedef struct{
   int *m;
}st;
void f1(st **pst){
   int i = 0;
   for(; i <9; i++){
      int j = *pst[i]->m;
   }
}
```

(a)

```
typedef struct{ int *a; }st;
int f2(st *ps){
   int *p = ps->a;
   *p = 2;
}
void f3(){
   st s;
      s.a = NULL;
   f2(&s);
}
```

(b)

Figure 2. Motivating examples

III. Region-Based Symbolic Three-Valued Logic

A. Ddressable Expression

Definition 4: *Memory Object.* The expression that corresponds to allocated memory when running programs, which can be top-level variable *v*, a member of a complex memory object, a dynamically allocated memory.

For all types of expressions defined by C99, we describe a C memory object expression *var* by the following grammar:

var::=v | *var.f* | *var*[*n*] | *malloc*(*exp*). Where *v* is top-level variable, *exp* is parameter.

Definition 5: *Addressable Expression.* The expression which has l-value and can be assigned.

For all types of expressions defined by C99, we describe a C addressable expression *aexp* by the following grammar:

aexp::= *var* | *aexp.f* | *aexp->f* | *aexp*[*exp*] | (*aexp*) | **aexp* | *id*(*exp*)

aexp* can be defined as: **aexp*::=aexp*' | *(++*aexp*') | *(--*aexp*') | *(*aexp*'++) | *(*aexp*' --) | *(*aexp*' *op exp*'), the type of *aexp*' is pointer, *op*= + | -, the type of *exp*' is integer.

For *id*(*exp*), where *id* is a method and return type is pointer, *exp* means parameters.

A memory object is an addressable expression, and there exist three relationships among addressable expressions as relationships between l-value and r-value.

- **Hierarchy**, relationship among l-values. It exists in addressable expression of compound type with its members.
- **Points-to relationship**, relationship of l-value and r-value. It exists in a pointer with the target that the pointer point to.
- **Linear and logical relationship**, relationship among r-values. The r-ralue of a memory unit has linear or logical relationship with the r-value of another memory unit.

Based on hierarchy and points-to relationships, we give the concept of parent addressable expression.

Definition 6: *Parent Addressable Expression*. Complex addressable expression is the parent of its members; Pointer is the parent of the addressable expressions that it points to.

For seven kinds of addressable expressions, *aexp i*s the parent of *aexp.f*, *aexp->f*, *aexp*[*exp*], **aexp*; *aexp->f* is equivalent to (**aexp*).*f*, whose parent is **aexp*.

When a function is called in different calling contexts, the points-to information of its pointer arguments maybe different. In order to map the points-to information at call site to the called function, we introduce extended variables to represent the points-to information of pointer parameters and global variables.

Let *e* as the variable that need to be extended, the extended rules are as follows:

- if *e* is a pointer, and the maximum level of dereference from *e* is *n*, then we create *n* extend variables, include **e*, ***e*,…and so on;
- if *e* is a variable with compound type and has *n* member, then we create *n* extend variables, include $e.f_1$, $e.f_2$,…and so on.

For example, parameter *ps* of function *f2* in Figure 2(b), the maximum level of dereference from *ps* is 1, so extended variable **ps* is introduced. **ps* is an extended variable and its type is struct and has a child *a*, so we generate an extended variable (**ps*).*a*. And (**ps*).*a* is pointer and the maximum level of dereference from it is 1, so we generate extended variable *(**ps*).*a*.

B. Rstvl

Definition 7: *Region-based Symbolic Three-Valued Logic*. RSTVL is a model of quadruple <*Var*, *Region*, S_{Exp}, *Domain*>, where *Var* is memory object, *Region* is abstract memory, S_{Exp} is symbolic expression, and *Domain* is the domain of value.

Quadruple RSTVL describes scalar memory object, and complex memory object can be decomposed into combination of scalar elements. Complex type memory object can be described by triple <*Var*, *Region*, *x*>, where *x* is determined by the type of *Var*, if the type of *Var* is array, *x* is {<*i*, *Region*>}, $i \in N$, *i* is the index of array *Var*; if the type of *Var* is struct, *x* is {< *f*, *Region*>}, *f* is the member of struct *Var*.

For different types of memory objects, different types of regions are applied. *PrimitiveRegion* describes primitive type memory object, *PointerRegion* describes pointer, *ArrayRegion* describes array, and *StructRegion* describes struct.

Each region has the only number, the numbering form of *PrimitiveRegion* is *bm_i* ($i \in N$), the numbering form of *PointerRegion* is *pm_i*, the numbering form of *ArrayRegion* is *am_i*, and the numbering form of *StructRegion* is *sm_i*. For the region dynamically allocated memory, its number is *mxm_i_n*(*x* means the type of the region, the value is '*b*', '*p*', '*a*' or '*s*'), *n* is bytes of memory size. The number of null address is "*null*", and the number of wild address is "*wild*". If the initial letter of the number of a region is '*u*' or '*g*', this region describes a parameter or global variable.

We call the region that maps *v*, *var.f*, *var*[*n*] is safe region, dynamically allocated region is dynamic region, the region that maps parameter or global variable is unknown region, these three kinds regions collectively call operable region; the region identified null or wild is an inoperable region. Dynamic region and unknown region will become safe region after not null judgement, dynamic region and unknown region will become inoperable region after is null judgement.

We divide domain [11] into two types: numeric and pointer, and apply *PointTos* to describe points-set in pointer domain *PointerDomain*, the elements of *PointTos* is the number of a region.

Domains of RSTVL and operators to them constitute complete lattice $< L, \leq, \sqcup, \sqcap, \bot, \top >$. $\bot$ is empty set; $\top$ of numeric domain is $[-\infty, +\infty]$; $\top$ of pointer domain is the union of null, wild and all numbers of operable region; $\sqcup$ is merge operation of sets; $\sqcap$ is intersection operation of sets. Static data flow analysis based on RSTVL can be transferred to operation on lattice.

RSTVL describes all three associations among addressable expressions; and is suitable for flow-sensitive, field-sensitive, context-sensitive and path-insensitive static analysis. Given a program point, a region abstraction based on RSTVL consists of the following:

- At each program point *l*, a set of regions R^l that models the locations that may access at *l*, a set S^l expresses symbols that may be used at *l*.
- At each program point *l*, exists an abstract store: $\rho^l = (\rho_v^l, \rho_r^l, \rho_f^l)$, where $\rho_v^l : V \to R^l$ maps memory objects to their regions; $\rho_r^l : R^l \to R^l$ expresses the points-to relationship among regions; $\rho_f^l : (R^l \times F) \to R^l$ maps members of a complex addressable expression to their regions.

To analyse an addressable expression, we need to get potentially associated regions first, and an addressable expression may associates several regions. At a program point *l*, if the abstract store is ρ, we use $R^l[\![e]\!]$ to express region set that addressable expression *e* associated. Then strategies can be given for achieving region set that all kinds of addressable expressions associated.

- $R^l[\![v]\!] = \rho_v^l(v)$;
- $R^l[\![e.f]\!] = \bigcup_{r \in R^l[e]} \rho_f^l(r, f)$;
- $R^l[\![e[i]]\!] = \bigcup_{r \in R^l[e]} \rho_f^l(r, i)$;
- $R^l[\![*e]\!] = \bigcup_{r \in R^l[e]} \rho_r^l(r)$;
- $R^l[\![*e]\!] = \bigcup_{r \in R^l[e]} \rho_r^l(r)$;
- $R^l[\![(e)]\!] = R^l[\![e]\!]$;
- $R^l[\![e->f]\!] = \bigcup_{r \in R^l[e]} \left\{ \bigcup_{r' \in \rho_r^l(r)} \rho_f^l(r', f) \right\}$.

We have applied RSTVL to data flow analysis in DTSGCC [9]; the analysis is flow-sensitive, field-sensitive, and context-sensitive based on symbolic

function summary, it can analyzes the over-approximation of every memory objects in every program point.

IV. Identifying Detecting Objects of Intraprocedural Null Dereference

A. Identifying Referenced Pointers

Based on the grammar defined by BNF, we generate AST for the C file under test. At the generating scope table stage, we identify addressable expressions from AST, and bind each addressable expression to the related node of AST [8].

There are three kinds of nodes that are closely related to addressable expressions, which are UnaryExpression, PostfixExpression and PrimaryExpression; and their grammars are described by BNF as follows:

UnaryExpression::= PostfixExpression | “++” UnaryExpression | “--” UnaryExpression | <SIZEOF> (UnaryExpression | “(“TypeName”) ”) | UnaryOperator CastExpression, UnaryOperator::= “&” | “*” | “+” | “-” | “~” | “! ”);

PostfixExpression ::= PrimaryExpression (“. ” <IDENTIFIER> | “[“Expression”] ” | “(“(ArgumentExpressionList)? ”) ” | “->” <IDENTIFIER> | “++” | “--”)*;

PrimaryExpression ::= <IDENTIFIER> |“(“Expression”)”| Constant.

According to grammatical features, pointer expression e_p as a kind of addressable expression can be divided in to three types: $*e_p$, e_p->f and $e_p[exp]$, so we can identify all dereference expressions from searing AST, and identified all referenced pointers. We apply XPath to search AST, and the query statement of $*e_p$ is:

.//AssignmentExpression//UnaryExpression[/UnaryOperator[@Operators='*']]/UnaryExpression.

The query statement of e_p->f and $e_p[exp]$ is:

.//AssignmentExpression//UnaryExpression/PostfixExpression[./PrimaryExpression][contains(@Operators,'[')or contains(@Operators, '->')].

For the example of Figure2(a) , $*pst[i]$->m is a $*e_p$ type pointer expression, so we can identify it from the related UnaryExpression node of AST, and deduce the pointer being referenced is $pst[i]$->m; $pst[i]$->m is a e_p->f type pointer expression, so we can identify it from the related PostfixExpression node of AST, and deduce the pointer being referenced is $pst[i]$; $pst[i]$ is a $e_p[exp]$ type pointer expression, so we can identify it from the related PostfixExpression node of AST, and deduce the pointer being referenced is pst. Above all, we identify three referenced pointers from $*pst[i]$->m: $pst[i]$->m, $pst[i]$, pst.

B. Points-to Attribute

We can decide whether a pointer being referenced is null dereference or not based on its points-to attribute. Points-to attribute is described as a lattice: $AL_{PTR} = (V_{PTR}, F_{join}, F_{meet})$, and its Hesse table is shown in Figure 3. V_{PTR} depicts the value set of points-to attribute, which can describe security of a pointer being referenced effectively, and can be conveniently applied to null dereference detection. EMPTY expresses initial value of attribute lattice, NULL expresses a pointer points to null address, NOTNULL expressed a pointer points to a safe memory address, NON (NULL_OR_NOTNULL) expresses a pointer may be points to null address. When a pointer is referenced, null dereference will inevitably occur if points-to attribute of the pointer is NULL, may occur if points-to attribute of the pointer is NON.

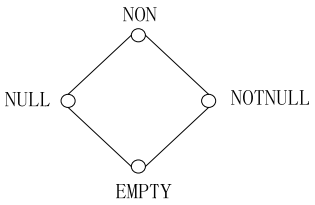

Figure 3. Hasse table of AL_{PTR}

F_{join}: $V_{PTR} \times V_{PTR} \rightarrow V_{PTR}$ is the greatest lower bound function of AL_{PTR}.

F_{meet}: $V_{PTR} \times V_{PTR} \rightarrow V_{PTR}$ is the least upper bound function of AL_{PTR}.

In order to comprehensive express points-to information, UNKNOWN is introduced to express uncertainty of points-to of a pointer; it is applied to initialize points-to attribute of pointer parameters and global variables. Operations about UNKNOWN with other attribute value X as follows:

F_{join}(X, UNKNOWN) = X

F_{meet}(NOTNULL, UNKNOWN) = UNKNOWN

F_{meet}(NULL, UNKNOWN) = NON

F_{meet}(NON, UNKNOWN) = NON

F_{meet}(EMPTY, UNKNOWN) = UNKNOWN

F_{meet}(UNKNOWN, UNKNOWN) = UNKNOWN

At program point l, let $T^l[\![r_{name}]\!]$ express the type of the region numbered r_{name}, pd express the domain of pointer e_p, the abstraction function α_ρ^l of points-to attribute is defined as follows:

$$\alpha_\rho^l(pd) = \begin{cases} \text{EMPTY} & pd = \varnothing \\ \text{NULL} & pd = \{\text{null}\} \\ \text{NOTNULL} & \forall pt \in pd, T^l[\![pt]\!] \text{ is safe} \\ \text{UNKNOWN} & (\forall pt \in pd, T^l[\![pt]\!] \text{ is safe or unknown}) \text{ and } (\exists pt \in pd, T^l[\![pt]\!] \text{ is unknown}) \\ \text{NON} & others \end{cases}$$

V. Identifying Detecting Objects Of Interprocedural Null Dereference

A. Function Summary

Each function call might affect its concrete call site context in four aspects:

- the callee function might cause side effects to actual-parameters and global variables;
- the caller’s dataflow and control flow might be transformed by callee’s return value;

- potential interrupt instructions, such as exit, assert, exception, etc;
- pre-condition, the call site context must obey the callee's invocation constraints to avoid defects.

In this paper, our function summary only focuses on pre-condition of null dereference NPDPreSummary. For the example in Figure 2(b), pointer parameter *ps* is referenced at line 3, and the points-to attribute of *ps* is UNKNOWN, so we must add *ps* can not be null as a pre-condition of function *f4*.

B. Generating Pre-condition of Null Dereference

If $\alpha_\rho(V^l[\![e_p]\!]) == \text{UNKNOWN}$ and the *PointerDomain* of e_p is *pd*; then for each region number r_{Name} in *pd* and the region named r_{Name} is an unknown region, if the momory object mapping to the region is *exp*, then we set the parent addressable expression of *exp* can not be null as pre-condition. Let $R_n^l[\![r_{name}]\!]$ express the region numbered r_{name} at program point *l*; Let $E_r[\![r]\!]$ express the memory object that related to *r*. The generating pre-condition of null dereference is detailed in algorithm 1.

Algorithm 1 Generating pre-condition of null dereference

Input: pointer being referenced e_p, NPDPreSummary
Output: NPDPreSummary
Declare: *getParent*(*para*): get father addressable expression of *para*.

for each $pt \in V^l[\![e_p]\!]$ && $T^l[\![pt]\!]$ is unknown region
  let $var = E_r[\![R_n^l[\![pt]\!]]\!]$;
  let *fvar* = *getParent*(*var*);
  add *fvar* to NPDPreSummary;
end for
return NPDPreSummary;

C. Instantiating Pre-condition of Null Dereference

For each function call, we get its function summary first, and instantiate the function summary based on the calling context at the call site.

Function call expression is a kinds of addressable expression, the grammar of it is *id*(*exp*). *id*(*exp*) maps to PrimaryExpression, it can be identified by searching AST, the query statement is:

.//PrimaryExpression[@Method='true']

If the called function has function summary, and the pre-condition constraint some pointers can not be null, then we instantiate it.

To instantiate the pre-condition of null dereference, the key is for each constrained pointer e_{cp} in pre-condition, get related addressable expression set e_p*List* at the call site; and based on the abstract store state at the call site described by RSTVL, get the points-to attribute for each pointer of e_p*List*. If the points-to attribute of a pointer in e_p*List* is UNKNOWN, then we add this pointer into pre-condition of null dereference applying algorithm 1, otherwise , the pointer is a detecting object of null dereference.

In all of above steps, the key is getting the addressable expression set for each constrained pointer in pre-condition of null dereference, which is a problem that maps a parameter to arguments; the details is shown in algorithm 2.

Algorithm 2 Mapping a Parameter to Arguments.
Input: *para*, R^n
Output: *VarsList*<Variable>
Declare:
getParents(*para*): get parent addressable expressions of *para* sorted according to parent-child relationship.
getArgument(*var*, *n*): get the corresponding arguments of top-level parameter *var* at the calling point *n*.
getParent(*var*): get parent addressable expression of *var*.
getType(*e*): get the addressable expression type of *e*,where 0: *v*, 1: *e.f*, 2: *e->f*, 3: *e*[*exp*], 4: (*e*), 5: **e*, 6: *m*(*exp*).
getMemName(*s*, *var*): for addressable expression *s* and the type of *s* is struct, get the member name of its child addressable expression *var*.

let *args*<Variable> = ∅;
let *parents*<Variable> = *getParents*(*para*);
get first variable v_0 in *parents*;
args = { *getArgument*(v_0, *n*) };
for each $p \in$ *parents* && $p \neq v_0$
  let v_p = *getParent*(*p*);
  let *vars*<Variable> = ∅;
  for each $v \in$ *args*
    for each $r \in R^{\bullet n}[\![v]\!]$
      if *getExpType*(*var*) == 1 then
        let *m* = getMemName(v_p, *p*);
        vars ∪= { $R_f^l[\![r, m]\!]$ };
      else **if** *getExpType*(*var*) == 5 **then**
        vars ∪= { $V_r^{\bullet n}[\![r]\!]$ };
      end if
    end for
  end for
args = *vars*;
end for
return *args*;

For the function *f2* in Figure2 (b), parameter *ps* is referenced at line 3; dereference *p* at line 4 is actually access the region that pointed by (**ps*).*a*. the points-to attribute of *ps* and (**ps*).*a* are UNKNOWN, so the pre-condition of null dereference of *f2* is: {*ps*[NOTNULL], (**ps*).*a* [NOTNULL]}.

Function *f3* calls *f2* at line 9, *ps* is a top-level parameter and constrained can not be null in pre-condition of null dereference, based on the numerical order, we can deduce *ps* maps to *&s* at call site, it's a safe dereference. (**ps*).*a* is also constrained can not be null in pre-condition, its parent addressable expression set is {*ps*, **ps*, (**ps*).*a*}. Based on the abstract store state at the call site described by RSTVL at line 9, we can deduce that *ps* maps {*&s*}, **ps* maps {*s*},

(**ps*).*a* maps {*s.a*}. So detecting objects of null dereference is *&s*, *s.a*, and the points-to attribute of *s.a* is NULL, so a null dereference defect will be reported at line 9.

VI. Experimental Analysis

We choose five C projects to validate the effectiveness of our approach.

A. Identifying Defecting Objects of Null Dereference

Pointers being referenced of five C projects identified by our approach are shown in TABLE I. Pointers being referenced can be divided as: local pointer and points-to attribute is known (LKP), local pointer and points-to attribute is unknown (LUP), external pointer and points-to attribute is known (EKP), external pointer and points-to attribute is unknown (EUP), function pointer (FP).

TABLE I
STATISTICS OF REFERENCED POINTERS

Benchmark	KLOC	Referenced pointer					
		LKP	***LUP***	***EKP***	***EUP***	***FP***	***Total***
antiword-0.37	24.2	770	142	50	1716	73	2751
uucp-1.07	52.6	1849	1112	401	2397	379	6138
sphinxbase-0.3	22.5	691	203	602	2011	82	3589
optipng-0.6	27	415	239	178	1258	53	2143
barcode-0.98	3.4	176	52	249	378	18	873
Total	130	3901	1748	1480	7760	605	15494

It is shown in TABLE I that pointer dereference occur frequently in C functions, about 120/KLOC, and more than 60% pointers being referenced can not be determined their points-to attribute in respective function, if function pointers are considered, more than 65% pointers being referenced need interprocedural identified.

For lib functions, we construct their function summary artificially. And detecting objects of null dereference identified by function summary can be divided two kinds: identified based on custom function (CFP), identified based on lib function precondition pointer (LFP). For pointers being referenced that can not be determined points-to attribute in TABLE I, we identify their relative detecting objects of null dereference by interprocedural identifying approach, the result is shown in TABLE II.

TABLE II
STATISTICS OF DETECTING OBJECTS OF NULL DEREFERENCE

Benchmark	Detecting objects of null dereference					
	LKP	***EKP***	***FP***	***CFP***	***LFP***	***Total***
antiword-0.37	770	50	73	372	105	1370
uucp-1.07	1849	401	379	1382	1314	5325
sphinxbase-0.3	691	602	82	170	222	1767
optipng-0.6	415	178	53	579	358	1583
barcode-0.98	176	249	18	56	259	758
Total	3901	1480	605	2559	2258	10803

For the example in Figure 1(b), *ps* is a EUP type referenced pointer, *p* is a LUP type referenced pointer, *s.a* is a CFP type detecting object of null dereference.

Applying our approach, more detecting objects of null dereference can be identified, and more null dereference defects can be detected. There is a null dereference in Figure 4, *Barcode_128_make_array* calls lib function *strlen* at line 321, the pre-condition of null dereference in function summary of *strlen* constrain its parameter can be null, and the points-to attribute of *bc->ascii* is UNKNOWN at line 321, so we add (**bc*).*ascii*(equals to *bc->ascii*) can not be null pointer into pre-condition of null dereference of *Barcode_128_make_array*. *Barcode_128_encode* calls *Barcode_128_make_array* at line 439, we can deduce that parameter (**bc*).*ascii* maps argument *bc->ascii*, and the points-to attribute of *bc->ascii* at line 439 is NON, so we make as *bc->ascii* a detecting object of null dereference, and report *bc->ascii* is a null dereference defect at line 439. Klocwork9 [12] and Saturn [13] can not detect these defects.

File:barcode\code128.c
The called functions at line 314:
static int *Barcode_128_make_array(struct Barcode_Item *bc, *)
321: len = 2 * strlen(bc->ascii) + 5;

The calle function at line 414:
int Barcode_128_encode(struct Barcode_Item *bc)
433: text = bc->ascii;
434: if (!text) {
……;
}
439: codes = Barcode_128_make_array(bc, &len); //NPD

Figure 4. A detecting object of null dereference identified by our approach

VII. Related Work

There is some research in the area of expression recognition that related to our work. Maksim O et al. [14] present core expression as canonical representation, they also identify expression from AST, but can not guarantee to identify all addressable expressions; so their method can not guarantee identify all referenced pointers. PenAnalysis [15] applies expression tree to representation expression, which is more complex than our method. S. Blazy et al.

In order to analysis expressions comprehensively, relations among expressions must be considered, otherwise the result will be inaccurate. Alias set and points-to set only focus on alias relationship, can not express hierarchy of compound variables; applying them can't analyze complex pointers effectively. As a region model, RSTVL is similar to Brian Hackett's memory model [16], and appropriated for shape analysis.

Specific to null dereference testing, it is an important work to identify detecting objects of null dereference. Although null dereference testing has been extensively studied, only few of past researches mention how to fully identify detecting objects. PSE [3] defines a simple pointer language, and regulated source code patterns for the null dereference property; but it can't guarantee identifying multilevel pointers effectively, especially interprocedural multilevel pointers. B. Cheng, etc apply access path for interprocedural pointer analysis [17], their access path is similar to our parent addressable expression, and they also use function summary.

VIII. CONCLUSIONS

In this paper, we introduce an approach for identifying detecting objects of null dereference. RSTVL describes abstract storage of each program point and relationships of addressable expressions, uses region number set to express pointer point to. Based on the correspondence between addressable expressions and nodes of AST, we identify all pointers being referenced from AST based on the grammar of pointer expression. If the points-to attribute of a pointer being referenced is can be determined, then we add the related pointer can not be null into the pre-condition of null dereference in function summary, and identified related detecting objects of null dereference at call site based on the abstract storage described by RSTVL.

REFERENCES

[1] Michael D, Graham Z, Samuel Z, "Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses, " In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, 2010, pp. 13-24.

[2] J. Huang, M. Bond. "Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management," In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. 2013, pp. 53-72.

[3] R. Manevich, M. Sridharan, S. Adams, "PSE: Explainint program failure via psotmortem static analysis, " In *Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering*, 2004, pp. 63-72.

[4] X. Ma, J. Wang, D. Wang. "Computing must and may alias to detect null pointer dereference, " *Leveraging Applications of Formal Methods, Verification and Validation*, 17(17): 252-261, 2008.

[5] M. Buss. Summary-based pointer analysis framework for modular bug finding [D]. Columbia: Columbia University, 2008

[6] Y. Xie, A. Aiken. "Saturn: A scalable framework for error detection using Boolean satisfiability," *ACM Transactions on Programming Languages and Systems*, 29(3): 1-43, 2007.

[7] M. Ravichandhran, K. Raghavan. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1033-1052. ACM, 2011.

[8] Y. Dong, Y. Xing, D. Jin, Y. Gong. "An approach to fully recognizing addressable expression," In *The 13th International Conference on Quality Software*, 2013, pp. 149-152.

[9] Y. Dong, D. Jin, Y. Gong, Y. Xing. "Static analysis of C programs via region-based memory model," *Journal of Software*, 25(2): 357-372, 2014 (in Chinese with English abstract).

[10] Y. Dong, D. Jin, Y. Gong. "Symbolic procedure summary using region-based symbolic three-valued logic," *Journal of Computers*, 9(3): 774-780, 2014.

[11] Y. Wang, Y. Gong, Q. Xiao, Z. Yang. "A Method of Variable Range Analysis Based on Abstract Interpretation and Its Applications," *Acta Electronica Sinica*, 39(2): 296-303, 2011 (in Chinese with English abstract).

[12] M. Webster. "Leveraging static analysis for a multidimensional view of software quality and security: Klocwork's solution," White paper, IDC. 2005.

[13] I. Dillig, T. Dillig, A. Aiken. "Sound, complete and scalable path-sensitive analysis," *ACM SIGPLAN Notices*, 43(6): 270-280, 2008.

[14] M. Orlovich and R. Rugina. "Core expressions: An intermediate representation for expressions in C," *In Submitted to Compiler Construction'06*. Available at http://www.cs.cornell.edu/~rugina.

[15] M. Strout, J. Mellor-Crummey, P. Hovland. "Representation-independent program analysis," In *Proceedings of the The sixth ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2005, pp. 64-74.

[16] B. Hackett, R. Rugina. "Region-Based Shape Analysis with Tracked Locations," In *Proceedings of the 32nd ACM SIGPLAN- SIGACT symposium on Principles of programming languages*, 2005, pp. 310-323.

[17] B. Cheng, W. Hwu. "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," *Acm Sigplan Notices*, 35(5): 57-69, 2000.

Yukun Dong, received his PhD in computer science from School of Beijing University of Posts and Telecommunications, Beijing, China, in 2014. He currently serves as a lecturer in College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China. His research interests include software testing and program static analysis.