Symbolic Procedure Summary Using Region-based Symbolic Three-valued Logic

Yukun Dong
State Key Laboratory of Networking and Switching Tech, Beijing University of Posts and Telecommunications
Beijing 100876, China
College of Computer and Communication Engineering, China University of Petroleum, Qingdao, China
Email: dongyk@upc.edu.cn

Dahai Jin Yunzhan Gong
State Key Laboratory of Networking and Switching Tech, Beijing University of Posts and Telecommunications
Beijing 100876, China
Email: {jindh, gongyz}@bupt.edu.cn

***Abstract*—One of the bottlenecks in interprocedural analysis is the difficulty in handling complex parameters. This paper proposes a novel approach to solve this problem: symbolic procedure summary, which is constructed using region-based symbolic three-valued logic (RSTVL). RSTVL is a memory model that can describe memory state of variables and all kinds of associations among them. Based on the result of intraprocedural analysis, we construct symbolic procedure summary described by RSTVL, and instantiate it at call site using the calling context. Our approach improves analysis precision as it achieves context-sensitive and field-sensitive interprocedural analysis. We apply this approach in defect detection, and experimental results show that it can effectively reduce both false negatives and false positives of defect detecting, and improve test accuracy at the same time.**

***Index Terms*—static analysis, procedure summary, inter-procedure analysis, context sensitive, field sensitive**

I. INTRODUCTION

Interprocedural analysis is a well-understood static analysis technique [1] since procedure call is popular in programs [2]. An effective method for interprocedural analysis is computing procedure summary [3-9], which can be thought of as some succinct representation of the behavior of the procedure that is also parametrized by any information about its input variables. Each procedure is analyzed once to construct its procedure summary; and once a procedure is called, it will be substituted by its summary at call site. Procedure summary improves inter-procedural analysis performance because it can avoid analysis behaviour of the called procedure repeatedly.

Different kinds of procedure summaries have been proposed in the past, including input-output summary table [4], summary represented by predict constraints [5], symbolic procedure summary [7] and transfer function [9]. Different procedure summary represents different aspects, there is no general way to represent or construct it. Of all the procedure summaries, symbolic procedure summary is convenient for context-sensitive analysis; context-sensitive analysis considers concrete context of call site, and results of call to the same procedure varies with different calling contexts. Symbolic procedure summary abstracts from concrete information to symbol and instantiates symbol at a call site.

One key of symbolic procedure summary is the representation of summary information, such as BDD [9] and STVL [7] are used. No matter which memory model is choice, if complex structure and associate of variables not considered comprehensively, the precision will be decline; because pointer, struct and array exist in C program, which cause alias, hierarchical, logic relationships exist among variables.

We use the example in Figure 1 to illustrate some obstacles of interprocedural analysis. Function *f1* accepts two parameters *p* and *q*, and sets pointer **q* to pointer *p->a* in line L3. Since *p->a* and **q* are parameters, arguments will be changed when *f1* is called. Precise interprocedural analysis needs to deduce which arguments corresponds to *p->a* and **q* respectively, and which arguments are updated as *p->a* is assigned.

```
L1:  typedef struct{   int *a;  }S;
L2:  int * f1(S *p, int **q){
L3:      p->a = *q;
L4:      return **q+1;
L5:  }
L6:  void f2(int i){
L7:       int b = 10, c = 100, *r;
L8:       S s1, *s2;
L9:      s2 = &s1;
L10:     if(i>0)   r = &b;
L11:     else       r = &c;
L12:     i = f1(s2, &r);
L13: }
```

Figure 1. Motivating examples

Manuscript received September 10, 2013; revised October 22, 2013; accepted October 26, 2013.
Corresponding author: Yukun Dong (dongyk@upc.edu.cn).

doi:10.4304/jcp.9.3.774-780

Once complex parameters are taken into account, field-sensitive interprocedural analysis is needed; that means we must abstract memory with a model and describe associations among variables. Many memory models have been proposed, such as big array model, shape graph, TVLA [10], point-to diagram, region-based memory model [11, 12], STVL [13], etc; unfortunately, these memory models only consider several relationships. TVAL and region-based memory model do not consider logic value association, while STVL does not consider hierarchy.

To solve the above problems, we propose region-based symbolic three-valued logic (RSTVL); it can describe memory state of any memory object and all kinds of associations. In order to achieve context-sensitive and field-sensitive interprocedural analysis, this paper introduces a symbolic procedure summary based on RSTVL (SPS_{RSTVL}).

This paper makes the following contributions:

- **RSTVL:** It proposes a novel memory abstraction that can describe all kinds of associations of variables.
- **SPS_{RSTVL}:** It gives a new symbolic procedure summary based on RSTVL, which is suitable to context-sensitive and field-sensitive interprocedure analysis.
- **DTSGCC:** It presents an automated defect testing system called DTSGCC, which is developed based on RSTVL and SPS_{RSTVL} that detect defects.

The paper is organized as follows. Section II introduces addressable expression and RSTVL. Section III describes the algorithm for computing symbolic procedure summary. Section IV presents how to instantiate symbolic procedure summary. We present experimental results in Section V, conclusion in Section VI.

II. Region-based Symbolic Three-Valued Logic

A. *Addressable Expression*[14]

The C programming language standard classifies expressions into l-value and r-value. An expression's l-value is the memory location of its associated memory object, and an expression's r-value is the value associated with the memory object. Some expressions have both l-value and r-value, for example integer variable *a*, and array element *arr[0]*; some expressions only have r-value, for example *a*b*, and *c||d*; only array variable have l-value but no r-value.

Definition 1 *Addressable Expression.* Array or an expression that has l-value and can be assigned.

Some expressions have l-values, but can't be assigned, for example *p*+2, where *p* is a pointer.

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

aexp::= *id* | *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.

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 it 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 relationship, we give the concept of parent addressable expression.

Definition 2 *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*.

B. *RSTVL*

Of all memory models that proposed, Xu[11] takes hierarchy and points-to relationship into account, and proposes region-based memory model <*Var*, *Region*, *Value*>; but this model omits linear and logic relationship, and is not suitable for path-insensitive analysis. STVL[13] is a ternary model <*Var*, S_{Exp}, *Domain*>; it considers points-to relationship, linear and logic relationship, but does not consider hierarchy. Combining these two models, we propose RSTVL.

Definition 3 *Region-based Symbolic Three-Valued Logic*(*RSTVL*) is a model of quadruple <*Var*, *Region*, S_{Exp}, *Domain*>, *Var* is memory object, *Region* is abstract memory, S_{Exp} is symbolic expression, *Domain* is the domain of value.

Static analysis based on RSTVL is a proposition of three-valued logic, that is <*Var*, *Region*, S_{Exp}, *Domain*> → {0, 1, 1/2}, $3v\ell \to 0$ expresses the value of *Var* is an unknown domain, $3v\ell \to 1$ expresses the value of *Var* is a concrete domain, $3v\ell \to 1/2$ expresses the value of *Var* is a symbol expression.

Quadruple RSTVL describes scalar memory object, 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 *Ar-*

rayRegion 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.

Definition 4 *Symbolic Expression*. A symbolic expression S_{Exp} is composed by symbols through mathematical and logical operations; it can be defined recursively as follows:

$$\begin{cases} S_{Exp} \to Rel_{Exp} \mid \neg Rel_{Exp} \mid Rel_{Exp}\ \ell\ S_{Exp}, \quad \ell = \{\&\&, \|\} \\ Rel_{Exp} \to Math_{Exp} \mid Math_{Exp}\ \Re\ Rel_{Exp}, \Re = \{<, \le, >, \ge, ==, \ne\} \\ Math_{Exp} \to Term \mid Term \pm Math_{Exp} \\ Term \to Power \mid Power \times Term \mid Power \div Term \\ Power \to Factor \mid Factor^{Power} \\ Factor \to Constant \mid Symbol \mid (S_{Exp}) \end{cases}$$

Symbol expresses the symbolic value of an addressable expression, and corresponds to an addressable expression at a program point. We employ domain [15] to express the value of a symbol.

We divide domain 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.

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.
- The association of abstract store expressed by RSTVL is a quadruple $\sigma = (\sigma_r, \sigma_s, \sigma_f, \sigma_d)$. $\sigma_r : V \to R$ expresses the relationship of addressable expressions with their regions, it's a many-to-many association, means an addressable expression corresponds to several regions, and a region describes abstract store of several addressable expressions. $\sigma_s : R \to S_{Exp}$ maps regions to their symbol expressions. $\sigma_f : S_{Exp} \to S$ expresses the relationships of symbolic expressions and symbols, it's a one-to-many association, means a symbolic expression is composed by several symbols. $\sigma_d : S \to D$ maps symbols to domains, and each symbol has a domain.

Since RSTVL applies to path-insensitive analysis, an addressable expression may associates several regions. To analyse an addressable expression, we need to get potentially associated regions first. 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.

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

We define other relevant operations for RSTVL as follows:

$V_r^l[\![r]\!]$: gets the symbolic expression of region *r* at program point *l*.

$V_e^l[\![e]\!]$: gets the symbolic expression of addressable expression *e* at program point *l*, $V_e^l[\![e]\!] = \|_{r \in R^l[\![e]\!]} V_r^l[\![r]\!]$.

$D_s^l[\![s]\!]$: gets the domain of symbol *s* at program point *l*.

$D_{se}^l[\![s_{Exp}]\!]$: calculates the domain of symbolic expression s_{Exp} at program point *l* based on interval arithmetic.

$E_r[\![r]\!]$: gets the mapped addressable expression of *r*.

$E_s[\![s]\!]$: gets the mapped addressable expression of *s*.

Figure 2 shows the region association before L12 of the example in Figure 1.

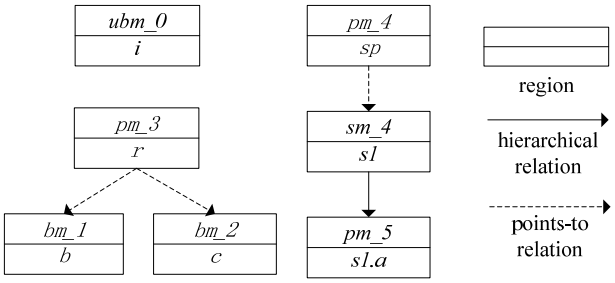

Figure 2. Region association diagram before L12

III. Constructing Symbolic Procedure Summary

Each procedure call might affect its concrete call site context in three aspects:

(1) The callee procedure might cause side effects to arguments and global variables;

(2) The caller's dataflow and control flow might be transformed by callee's return value;

(3) Potential interrupt instructions, such as exit, assert, exception, etc. This paper focuses on the first two aspects.

When a procedure is called in different calling contexts, the points-to information of pointer arguments and global variables maybe different. In order to map the points-to information at call site to the called procedure,

we introduce extended variables [16] to represent the points-to information of pointer parameters and global variables. The extended rules are as follows, if *p* is a pointer variable that needs to be extended, the maximum level of dereference from *p* is the number of extended variables for *p*.

For example, parameter *q* of procedure *f1* in Figure 1, the maximum level of dereference from *q* is 2, so extended variables * *q* and ** *q* are introduced. For complex parameters and global variables, we generate extended variables based on their members; for example, parameter *p* of procedure *f3* in Figure 1 is pointer, **p* is an extended variable and its type is struct and has a child *a*, so we generate an extended variable *(*p).a*. At the entry point of procedure *p*, we generate extended variable for all parameters and global variables that are used in *p*.

Definition 5 *Symbolic Procedure Summary Using RSTVL*(SPS_{RSTVL}). It describes the behaviour of a procedure based on RSTVL with four types of information:

$$\begin{cases} VarS_{Exp}Set = \{< Var, S_{Exp} >\} \\ RetS_{Exp} = S_{Exp} \\ SymbolDomainSet = \{< Symbol, Domain >\} \\ RegionVarSet = \{< Region_{Name}, Var >\} \end{cases}$$

We construct symbolic procedure summary for a procedure *p* in the following way:

First, at the entry point of *p*, we generate extended variables for parameters and global variables, and create regions for them;

Second, based on the analysis result of intraprocedural analysis for *p*, we calculate symbolic expression $RetS_{Exp}$ of return statements;

Third, for any variable *v* which is global variable or parmeter and their extended variables, if *v* is pointer and its symbolic expression s_{Exp} at exit point of *p* is different from its symbolic expressions at entry point of *p*, we add < *v*, s_{Exp} > to $VarS_{Exp}Set$;

Fourth, for any symbol *s* exists in symbolic procedure summary, we get its *domain* and add <*s*, *domain*> to *SymbolDomainSet*.

Fifth, for each <*symbol*, *domain* >, if the *domain* is *PointerDomain*, we get its points-to set *pts*, for each region number $region_{Name}$ in *pts*, we get its related variable *var*, and add < $region_{Name}$, *var*> to *RegionVarSet*;

Finally, add $VarS_{Exp}Set$, $RetS_{Exp}$, *SymbolDomainSet* and *RegionVarSet* to SPS_{RSTVL}. As an attribute, SPS_{RSTVL} is stored in the global environment.

For a procedure *p*, let G_{exit} be the exit point of *p*, G_{exit} be the last node of CFG(Control Flow Graph) of *p*. Algorithm 1 shows the algorithm of constructing SPS_{RSTVL}.

Algorithm 1 *Constructing* SPS_{RSTVL}

Input: G_{exit}

Output: SPS_{RSTVL}

1 $S_{Exp}List = \varnothing$; $RetS_{Exp} = \perp$; $VarS_{Exp}Set = \varnothing$; $SymbolDomainSet = \varnothing$; $RegionVarSet = \varnothing$;
2 **let** V_{Set} as the set of global variables, parameters and extended variables
3 **foreach** $n_{Ret} \in pre(G_{exit})$
4 $RetS_{Exp} = RetS_{Exp} \,||\, getS_{Exp}(n_{Ret})$;
5 $S_{Exp}List = \{ RetS_{Exp} \}$;
6 **foreach** $v \in V_{Set}$
7 **if** $V_e^{G_{exit}}[\![v]\!] \neq V_e^{G_{entry}}[\![v]\!]$ **then**
8 $VarS_{Exp}Set \cup = \{< v, V_e^{G_{exit}}[\![v]\!] >\}$;
9 $S_{Exp}List \cup = \{ V_e^{G_{exit}}[\![v]\!] \}$;
10 **foreach** $s \in S_{Exp}List$
11 $SymbolDomainSet \cup = \{< s,\ D_s^{G_{exit}}[\![s]\!] >\}$;
12 **if** $D_s^{G_{exit}}[\![s]\!]$ is *PointerDomain* **then**
13 **let** *ptList* as points-to set of $D_s^{G_{exit}}[\![s]\!]$;
14 **foreach** $r_{Name} \in ptList$
15 $RegionVarSet \cup = \{< r_{Name}\ ,\ E_r[\![R_n^{G_{exit}}[\![r_{Name}]\!]]\!] >\}$;
16 add $VarS_{Exp}Set$, $RetS_{Exp}$, *SymbolDomainSet*, *RegionVarSet* to SPS_{RSTVL};

For *f1* in Figure 1, its SPS_{RSTVL} includes:

- $VarS_{Exp}Set$: {<*q*, *q_67*>, <*p*, *p_01*>, <**q*, **q_89*>, <**p*, **p_1011*>, <*(*p).a*, *(*p).a_45*>, <(*p).a, (*p).a_23*>, <***q*, ***q_1213*>};
- *RegionVarSet*: {<"*upm_5*", **q*>, <"*usm_1*", **p*>, <"*ubm_6*", ***q*>};
- $RetS_{Exp}$: 1+****q_1415*;
- *SymbolDomainSet*: For symbols, *p_01*, *(*p).a_45*, *q_67*, **q_89*, their domain is *PointerDomain*, their points-to sets are {"*usm_1*"}, {"*ubm_6*"}, {"*upm_5*"}, {"*ubm_6*"} respectively.

IV. Instantiating Symbolic Procedure Summary

Suppose the CFG of procedure *p* is *G*, and *p* calls *q* at node *n*, expressed as $p \xrightarrow{n} q$. We give the basic process that initialize SPS_{RSTVL} of *q* in the following way:

First, we get the SPS_{RSTVL} of *q* at *n*;

Second, based on the relationship of parameters with arguments, we deduce the relationship of extended variables exist in SPS_{RSTVL} with addressable expressions in *p*; this step is described in Algorithm 2;

Third, for each *PointerDomain pd* in SPS_{RSTVL}, we get its points-to set and update it with region numbers at *n*, thus realize instantiating *pd*; this step is described in Algorithm 3;

Fourth, for each symbol *s* in any symbolic expression sym_{exp} of SPS_{RSTVL}, we get its corresponding variable *v*, and deduce variable set *vs* that *v* mapped based on the calling context; then combine all the symbolic expression of each variable in *vs* and get the symbolic expression s_{exp}, then replace *s* with s_{exp};

Fifth, we update active variables that influenced by a procedure calling side effect based on the instantiated symbol;

Finally, we calculate the returned symbolic expression based on the instantiated symbol. Algorithm 4 shows all the steps.

Algorithm 2 *MappingToArguments*, Mapping a Parameter to Arguments.

Input: *para*, R^n

Output: *VarsList*<Variable>

getParents(*para*): For *para*, get parent addressable expressions set Var_{List} ordered in parent-child relationship.

1 *VarsList* = $\varnothing$;
2 Var_{List}=*getParents*(*para*);
3 get first variable v_0 in Var_{List};

4 $ap_0 = getArgument(v_0)$;
5 **foreach** $var \in Var_{List}$ and $var \neq v_0$
6 $v_i = getParent(var)$;
7 $ListVarVi = \varnothing$;
8 **foreach** $region \in R^n[\![v_i]\!]$
9 $ListVarVi \cup = \{V_r^n[\![region]\!]\}$;
10 *VarsList = ListVarVi;*

For the example in Figure 1, *f2* calls *f1*, parent variables set of extended variable *p->a* is {*p*, **p*, *(*p).a*}, *p* is the first parameter and corresponds to argument *s2*, **ps* corresponds to *s1*, thus *(*p).a* corresponds to *s1.a*.

Algorithm 3 *InitializingPointerDomain*, Initializing a *PointerDomain* based on RSTVL.

Input: PointerDomain *pd*

Output: PointerDomain *pd*

1 *pts = getPointsTo(pd)*;
2 **foreach** *pt* ∈ *pts*
3 **if** *pt* ≠ "null" and *pt* ≠ "wild" **then**
4 *var = getVar(pt, RegionVarSet)*;
5 remove *pt* from *pts*;
6 **if** *var* is a parameter or global variable **then**
7 *varList = getVars(var, VarsList)*;
8 **foreach** *v* ∈ *varList*
9 **foreach** $region \in R^n[\![v]\!]$
10 add the number of *region* to *pts*;
11 **else**
12 add "wild" to *pts*;

For the example in Figure 1, *f2* calls *f1* at line L12, the points-to set of the *PointerDomain* that symbol *(*p).a_45* associated is {"*ubm_6*"}. "*ubm_6*" is the region number of ***q*; and ***q* corresponds to arguments *b* and *c*; the region numbers of *b* and *c* are "*bm_1*" and "*bm_2*", thus the points-to set is updated to {"*bm_1*", "*bm_2*"}.

Algorithm 4 *Symbolic Procedure Summary Initializing Algorithm based on RSTVL*

Input: *method*

Output: $SExp_{Ret}$

1 *sps = getSummary(method)*;
2 get $VarS_{Exp}Set$, $RetS_{Exp}$, *SymbolDomainSet*, *RegionVarSet* from *sps*;
3 $fVarAVarsSet = \varnothing$;
4 **foreach** $< var, s_{Exp} > \in VarS_{Exp}Set$
5 *varsList = MappingToArguments (var)*;
6 $fVarAVarsSet \cup = varsList$;
7 **foreach** $< s, d > \in SymbolDomainSet$ and *d* is *PointerDomain*
8 *InitializingPointerDomain(d);*
9 **foreach** $< var_p, s_{Exp} > \in VarS_{Exp}Set$
10 $tempS_{Exp} = s_{Exp}$;
11 **foreach** $s \in S[\![s_{Exp}]\!]$
12 **let** $e = E_s[\![s]\!]$, $newS_{Exp} = \perp$;
13 **foreach** $r \in R^n[\![e]\!]$
14 $newS_{Exp} \cup = V_r^n[\![r]\!]$;
15 replace *s* in $tempS_{Exp}$ with $newS_{Exp}$;
16 $var_a s = fVarAVarsSet(fVarAVarsSet, var_p)$;
17 **if** $|R^n[\![var_a s]\!]| == 1$ **then**
18 *strongUpdate*($tempS_{Exp}$, var_a);
19 **else**
20 *weakUpdate*($tempS_{Exp}$, var_a);
21 **if** $SExp_{Ret} \neq \perp$ **then**
22 **foreach** $s \in S[\![SExp_{Ret}]\!]$
23 **let** $e = E_s[\![s]\!]$, $newS_{Exp} = \perp$;
24 **foreach** $r \in R^n[\![e]\!]$
25 $newS_{Exp} \cup = V_r^n[\![r]\!]$;
26 replace *s* in $SExp_{Ret}$ with $newS_{Exp}$;

For the procedure call at line L12 in Figure 1, the points-to set of the *PointerDomain* that symbol *(*p).a_45* associated is updated to {"*bm_1*", "*bm_2*"}. *(*p).a_45* is generated for *(*p).a*, and *(*p).a* maps argument *s1.a*. So we can deduce the points-to set of the *PointerDomain* that *s1.a* associated is updated to {"*bm_1*", "*bm_2*"}, that means pointer *s1.a* poins to *a* or *b* after *f1* calls *f2*.

V. Experimental Evaluation

To evaluate the efficiency of our technique, we have implemented a defect detection system for GCC programs called DTSGCC, a practical and fully automatic defect detector. In this section, we first present the experimental setup. Then, we analyze the results and give our experimental discussion.

A. DTSGCC

DTSGCC analyzes programs in five stages as shows in Figure 3. First, loading defect patterns that need to detected, DTSGCC supports more that a hundred defect patterns. Second, collecting all source files that need to be detected and other related files; applying the preprocessing function of GCC compiler, which includes replacing macro definitions, spreading head files and executing conditional compilation based on testing environment, creating an intermediate file for each source file. Third, based on dependencies of files, getting analysis order of intermediate files. Fourth, create analysis thread for each intermediate file. Finally, storing and statistics analysis result.

The fourth stage can be broken into seven steps, which analysis an intermediate file. First, generating AST(Abstract Syntax Tree)[17] for the intermediate file. Secod, recognizing symbol by traversing AST, and generating symbol table. Third, analysis function calls relationships for all procedure in the intermediate file. Fourth, generating CFG for each procedure. Fifth, creating def-use chains for all variables based on CFG. Sixth, data flow analysis for each procedure based on the reverse order of function call, analysis abstract storage for each variable at each program point, we summarize each procedure and instantiate the callee summaries at concrete call sites. Finally, detecting potential defects along the control flow graph, using those above data flow information.

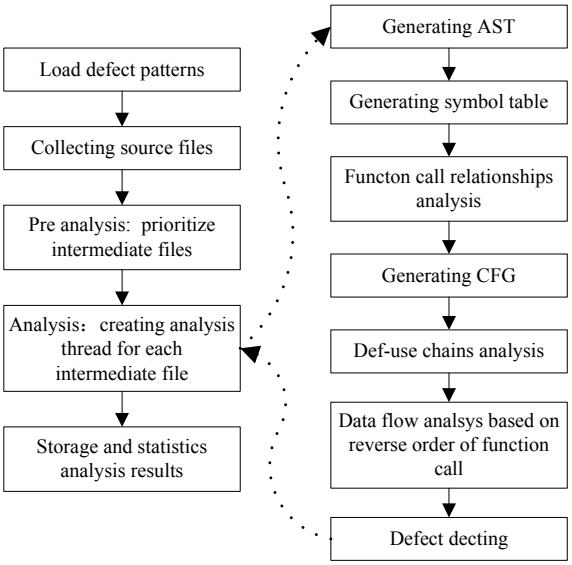

Figure 3. Analysis stages of DTSGCC

B. Experimental Setup

We choose five open source benchmarks to detect potential NPD defects. We believe all benchmarks to be challenging and interesting because they contain many complex struct and pointer usage patterns. All experiments were run on a dualprocessor 1.80GHz Pentium E2160 running Windows XP(SP3) with a 2GB physical memory. We measured running time using enough repetitions to avoid timer resolution errors.

To quantify the efficiency and precision, the experiments are conducted in two different configurations. The first configuration is based on STVL, which does not consider hierarchy of complex type. The second configuration is based on RSTVL.

TABLE I. gives the NPD defect number and analysis time of the various configurations, in which IP (Inspection Point) is detected by DTSGCC, BUG is validated by human.

TABLE I.
ANALYSIS RESULT OF C PROJECTS

C Project	Line	DTSGCC_STVL		DTSGCC_RSTVL	
		BUG/IP	*time(m)*	*BUG/IP*	*time(m)*
Antiword-0.37	25673	18/42	213	23/43	367
Barcode-0.98	8898	3/10	95	4/12	118
Uucp-1.07	52595	128/267	617	146/329	942
Sphinxbase-0.3	21138	47/141	248	58/162	283
Make3.81	23284	34/87	215	58/114	245
Stastistics	131588	230/547	1388	289/660	1955

C. Experimentation Analysis and Discussion

According to the statistics, 131KLOC were analyzed. All projects were analyzed in 23 minutes by DTSGCC_STVL with a false positive ratio of 58%, compared to 33 minutes by DTSGCC_RSTVL with a false positive ratio of 56%; DTSGCC_RSTVL detects 59 more bugs.

The main reason that makes DTSGCC_RSTVL spend more time is that it takes more associations among variables into account.

Figure 4 shows a representative NPD defect found by DTSGCC_RSTVL, which had not been detected by DTSGCC_STVL. The callee procedure may be assigned *pInfoCurrent->pBlockCurrent* a null pointer at line 530, so *pReadinfo->pBlockCurrent* may be null call site line 595, which causes the NPD defect at line 602.

```
File: antiword\blocklist.c
In callee procedure usGetNextByte(*, readinfo_type *pInfoCurrent, *,
*, *, *) at line 486
530: pInfoCurrent->pBlockCurrent = NULL;

In caller procedure usGetNextChar at line 557:
595: usLSB = usGetNextByte(pFile, pReadinfo, pAnchor, pulFileOff-
set, pulCharPos, pusPropMod);
602: if (pReadinfo->pBlockCurrent->tInfo.bUsesUnicode);  //NPD
```

Figure 4. A NPD defect detected by DTSGCC_RSTVL

VI. CONCLUSIONS

Interprocedural analysis is an important enabling technology. In order to describe procedure summary adequately, we proposed symbolic procedure summary using RSTVL, which can describe all kinds of associations of memory objects.

Our approach can handle complex parameters, and achieves field-sensitive and context-sensitive interprocedural analysis. Experimental results show that our technique can be applied to real-world programs and improve analysis precision.

The main inadequacy of our symbolic procedure summary is the ignorance of path condition, which might decrease the presion. In the near future, we will add path condition to symbolic procedure summary.

ACKNOWLEDGMENT

This work is supported by the Major Research plan of the National Natural Science Foundation of China under Grant 91318301, the High Technology Research and Development project in China (National 863 project in China) under Grant 2012AA011201, and the National Natural Science Foundation of China under Grant 61202080.

REFERENCES

[1] Fuad, Mohammad Muztaba, Debzani Deb, and Jinsuk Baek. “Static analysis, code transformation and runtime profiling for self-healing,” *Journal of Computers*, vol. 8, no.5, 2013, pp. 1127-1135.

[2] Z. Li, J. Tian. “A software behavior automaton model based on system call and context,” *Journal of Computers*, vol. 6, no.5, 2011, pp. 889-896.

[3] E. M. Nystrom, H.-S. Kim, and W. mei W. Hwu, “Bottom-up and top-down context-sensitive summary-based pointer analysis,” in *International Symposium on Static Analysis*, 2004, pp. 165-180.

[4] G. Yorsh, E. Yahav, S. Chandra, “Generating precise and concise procedure summaries,” *ACM SIGPLAN Notices*, vol.43, no.1, pp. 221-234, jan 2008..

[5] S. Gulwani, A. Tiwari. “Computing procedure summaries for interprocedural analysis,” in *Proceedings of*

the 16th European conference on Programming, 2007, pp. 253-267.
[6] I. Dillig, T. Dillig, A. Aiken, M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 567-577.
[7] Y. Zhao, L. Liu, Z. Yang, Q. Xiao, and Y. Gong, "Context-sensitive interprocedural defect detection based on a unified symbolic procedure summary model," in *Proceedings of the 2011 11th International Conference on Quality Software*, 2011, pp. 51-60.
[8] L. Shang, X. Xie, J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2004, pp. 264-274.
[9] J. Zhu, S. Calman, "Context sensitive symbolic pointer analysis," in *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, pp. 516-531, 2006.
[10] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1999, pp.105-118.
[11] Z. Xu, T. Kremenek and J. Zhang. "A memory model for static analysis of C programs," in *Leveraging Applications of Formal Methods, Verification, and Validation*, 2010, pp.535-548.
[12] 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.
[13] Y. Zhao, Y. Wang, *et al.*, "STVL: Improve the Precision of Static Defect Detection with Symbolic Three-Valued Logic," in *The 8th Asia-Pacific Software Engineering Conference*, 2011, pp.179-186.
[14] Y. Dong, Y. Xing, *et al.*, "An Approach to Fully Recognizing Addressable Expression," in *The 13th International Conference on Quality Software*, 2013, pp.149-152.
[15] Y. Wang, Y. Gong, *et al.*, "Variable range analysis based on interval computation," *Journal of Beijing University of Posts and Telecommunications*, vol. 32, no. 3, pp.36-41, 2009.
[16] B. Huang, *et al.*, "Context-Sensitive Interprocedural Pointer Analysis,"*Journal of Computer*, vol. 23, no. 5, pp.477- 485, 2000.
[17] Pattanayak, Binod Kumar, Sambit Kumar Patra, and Bhagabat Puthal, "Optimizing AST node for Java script compiler a lightweight interpreter for embedded device," *Journal of Computers*, vol. 8, no.2, 2013, pp. 349-355.

Yukun Dong, pursuing the Ph.D. degree in School of Beijing University of Posts and Telecommunications, Beijing, China; and 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.

Dahai Jin, received his PhD in computer science from Armored Force Engineering Institute, Beijing, China, in 2006. He currently serves as an associate professor in State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. His research interests include software testing and program static analysis.

Yunzhan Gong, received his PhD in computer science from Institute of computing technology, Academia Sinica, Beijing, China, in 1992. He currently serves as a professor in State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. His research interests include software testing, program static analysis and automatic test case generation.