Received December 8, 2017, accepted January 11, 2018, date of publication February 13, 2018, date of current version March 15, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2797314

RARE: An Efficient Static Fault Detection Framework for Definition-Use Faults in Large Programs

LUJIE ZHONG[1], PEN-CHUNG YEW[2], (Fellow, IEEE), WEI HUO[3], FENG LI[3], XIAOBING FENG[4], AND ZHAOQING ZHANG[4]

[1]Information Engineering College, Capital Normal University, Beijing 100048, China
[2]Department of Computer Science and Engineering, University of Minnesota at Twin Cities, Minneapolis, MN 55455, USA
[3]No. 6 Research Laboratory, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[4]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China

Corresponding author: Lujie Zhong (zhonglj@cnu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61402303 and in part by the Project of Beijing Municipal Commission of Education under Grant KM201510028016.

ABSTRACT A range-reduced static definition-use (def-use) fault detection framework is proposed to improve the scalability, but still retain its accuracy, when applied to large application programs. It casts common faults, such as null pointer dereferences, undefined references, buffer overflows, and memory leaks into a common def-use fault pattern, and uses a two-level path-insensitive approach to classify variable uses that can trigger faults into must-trigger, must-not-trigger, and may-trigger categories depending on whether the unsafe uses can actually be, never be, or may be executed. For those must-trigger unsafe uses, faults are immediately reported, and those must-not-trigger uses are dropped from further analysis. The already reduced program range that is relevant to the may-trigger unsafe uses is further reduced by using a binary decision diagram encoded path extraction scheme for more accurate, but more expensive, path-sensitive analysis. A prototype has been built using this approach, and a set of large realistic applications (a total of more than 4.8 MLOC) was tested for such common types of def-use faults. Compared with existing popular path-sensitive detection tools such as Clang Static Analyzer, we find our approach incurs less analysis time, but achieves good accuracy with a low false positive rate and no false negative.

INDEX TERMS Accuracy, fault detection, scalability, sensitivity, software reliability.

I. INTRODUCTION

Many common program faults, such as null pointer dereference (*NPD*), undefined reference (*UDR*), buffer overflow (*BO*), and memory leak (*ML*), are caused by unintended and/or unsafe definitions of certain variables. Such faults can lead to program vulnerability and even cause disastrous consequences. This kind of faults is commonly referred to as definition-use (*def-use*) faults because it is caused by assigning an unsafe value at a variable's definition site. As the assigned value flows through an execution path to the variable's use site, a fault will then be triggered. For a *def-use* fault, the use site is called a sink, and its corresponding definition site is called a source. Using static approaches, i.e. compile time approaches, to detect such faults accuracy and scalability is of primary concern when applied to very large application programs. However, achieving both accuracy and scalability is quite challenging as most existing techniques have shown.

For the accuracy-driven approaches, path-sensitive analysis is the primary technique used. They apply path-sensitive analysis inter-procedurally on the whole program, and solve the constraints along each control/execution path. The results are usually very accurate and sound. But for very large application programs with a large number of potential execution paths, their scalability suffers significantly. It may either take a very long time to complete the analysis, or simply fail to complete at all. In Saturn [1], an accuracy-driven approach using path-sensitive techniques, it infers boolean satisfiability [2] on each execution path to detect faults. It could suffer long analysis time, and has a limitation when applied to very large application programs in practice.

For those approaches that are more scalability-driven, on-demand strategies are often used to significantly reduce the analysis time. For example, Clang Static Analyzer (Clang-SA) [3] is a fault detection tool used as a plug-in to the Clang compiler. It restricts its analysis to certain types of condition expressions on branches for scalability concern. But it may misdiagnose some infeasible paths as feasible, and produce many false positive results. In another example, *Marple* [4] tries to limit path-sensitive analysis to only the execution paths that contain potential faults. Using this approach, they can exclude many execution paths that are irrelevant to faults. But it applies path-sensitive analysis to all other paths, which can still be a very large number for large application programs. Many of those remaining paths can actually be analyzed by much less expensive approaches equally effective in fault detection.

For example, values from definition sites that are passed along all paths of a branch tree down to a use site located below the confluence of the branch tree will always (i.e. must) trigger an unsafe use. In this case, much less expensive path-insensitive analysis can be equally effective and time saving, in particular for a very large branch tree. Hence, it is very important to distinguish among different program regions to apply either path-sensitive or path-insensitive analysis even after most irrelevant execution paths are excluded. We still need to avoid applying path-sensitive analysis as much as possible.

Another important consideration in fault detection is to adopt approaches that can fit well in existing compiler frameworks. Dataflow analysis is a general computation and propagation framework in most compilers. Instead of computing and propagating fault information outside of it, we should compute and propagate fault attribute (*FA*) values in it as much as possible. In our proposed approach, a fault attribute lattice (*FAL*) is introduced to facilitate this approach (see Fig.3 and Section II-C).

As more application programmers become more safety conscious, many build safeguards against potential faults in their programs. For example, they often check whether a pointer is *NULL* or not before dereferencing the pointer. This is usually done by guarding the pointer deference with some conditional statements with such a check. Hence, even though the pointer dereferencing could be unsafe (i.e. has a *NULL* value), but the fault will never be triggered. Similar cases exist in memory allocation, e.g. the program will return if a check shows that the memory allocation fails. The more robust the program is built, the more such safeguard statements exist. Interestingly, since such safeguard statements are placed by the programmers themselves, these safeguard statements have very limited and simple patterns that can be easily identified using simple pattern matching techniques. Existing def-use fault detection schemes often unnecessarily spend a lot of time analyzing such seemingly unsafe uses with little useful outcome, or report them as false positives.

To address both the accuracy and scalability problems, we propose a RARE (*RAnge-REduced*) framework with a two-level approach to select only a small subset of execution paths and program regions to apply expensive path-sensitive analysis (see Fig.2). In the first level, we use a less expensive path-insensitive but comprehensive (i.e. it is flow-, field-, and context-sensitive) approach to classify sinks into *safe*, *must-unsafe* and *may-unsafe* categories. Based on the execution path and control flow, a sink is further identified as *must-trigger*, *must-not-trigger* or *may-trigger* depending on whether the use will definitely be executed and trigger a fault, will definitely not be executed and will not trigger a fault, or may be executed and may trigger a fault, respectively.

For sinks with a safety attribute of safe, they can be dropped for further analysis because they can never trigger a fault. For a *must-unsafe* sink (e.g. a pointer with a *NULL* value), if it will definitely be executed (i.e. it is in the *must-trigger* category), a fault can be immediately reported. For those *may-unsafe* sinks, if they have safeguard statements they are in the *must-not-trigger* category, and they can also be discarded for further analysis (Fig.2).

For the remaining *may-unsafe* sinks that are in the *may-trigger* category further analysis is required. However, before performing path-sensitive analysis on these *may-trigger*sinks, we use a binary decision diagram (*BDD*) [5] scheme to encode path conditions in a compressed way, and a prefix *BDD* string-based path extraction scheme to further reduce the program range needed for path-sensitive analysis (see Section III-C). Such a two-level approach for def-use fault detection can be easily adopted in an existing compiler to take advantage of the most recent advances in program analysis.

In this paper, we made the following contributions.

- We propose a two-level static def-use fault detection framework. To improve accuracy, it employs both comprehensive path-insensitive analysis and range-reduced path-sensitive analysis in a unified framework. For better scalability, it uses a two-level approach to narrow the range to only a small subset of execution paths and program regions for applying path-sensitive analysis.
- We built a prototype in *Open64* [6] and experimented on a wide range of realistic programs (exceeds 4.8MLOC). It shows that our two-level strategy can reduce the scope of path-sensitive analysis to a much smaller range than existing approaches. Compared to other path-sensitive tools such as Clang-SA [1], our approach shows significantly lower false positives and fewer false negatives with faster analysis time.

The rest of the paper is organized as follows. Section II describes the def-use fault detection problems this paper intends to address. In Section III, our fault detection framework and its implementation are described in detail. Section IV shows the experimental environment, benchmarks, experimental data, fault detection results and some analysis. Section V discusses the related work. And finally, in Section VI, we conclude the paper.

II. PROBLEM STATEMENT

A. EXPLICIT AND NON-EXPLICIT DEF-USE FAULTS

The cause of all def-use faults stems from unsafe definitions early in the execution paths. To detect such faults, the uses of the variables that may trigger such faults need to be analyzed. We call the variables that are relevant to def-use faults characteristic variables. For example, in the detection of

```
1:  int *boo
    (int *x,int *y){
2:   int *q;
3:   if(*x>*y)
4:    q1=x;
5:   else
6:     q2=y;
7:   return *q;
8:  }
9:  void foo() {
10:  int *p1=NULL;
11:  int a,b,c;
     scanf("%d,%d,
12:     %d",&a,
        &b,&c);
13:  if (a>b)
14:   p2=&a;
15:  else if(b>c)
16:    p3=(int *)malloc
        (sizeof(int));
17:  else if(a>100)
18:    p4=boo(&a,&b);
19:  else
20:    p5=boo(p1,&b);
21:  c=*p+b;
22:  if(!p) return;
23:   *p=a*b;
24: }
25: void main() {
26:  foo();
27: }
```

(a)

(b)

DEF	Line	Fault Attribute Value	Safety Attribute
p_1	10	*NULL*	*Must Unsafe*
p_2	14	*NOTNULL*	*Safe*
p_3	16	*MAYNULL*	*May Unsafe*
p_4	18	*NOTNULL*	*Safe*
p_5	20	*MAYNULL*	*May Unsafe*
p_{phi}	21	*MAYNULL*	*May Unsafe*
p_{phi}	23	*NOTNULL*	*Safe*

(c)

Call Sites	Actual_Ins' Fault Attribute Values	Use_Ins' Fault Attribute Values	DEF	
line 18: boo()	*&a: NOTNULL* *&b: NOTNULL*	*x: NOTNULL* *y: NOTNULL*	q_1	*NOTNULL*
			q_2	*NOTNULL*
			q_{phi}	*NOTNULL*
line 20: boo()	*p₁: NULL* *&b: NOTNULL*	*x: NULL* *y: NOTNULL*	q_1	*NULL*
			q_2	*NOTNULL*
			q_{phi}	*MAYNULL*

FIGURE 1. An example of intra- and inter-procedural fault attribute value computation. (a) Example program. (b) Intra-procedural FA value computation.

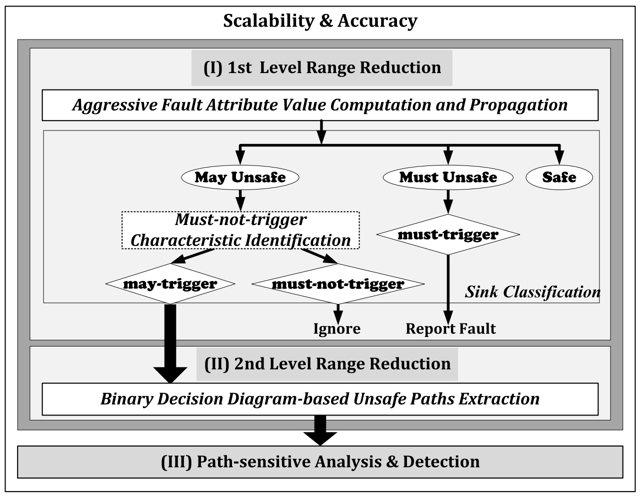
Scalability & Accuracy
(I) 1st Level Range Reduction
Aggressive Fault Attribute Value Computation and Propagation
May Unsafe
Must Unsafe
Safe
Must-not-trigger Characteristic Identification
must-trigger
may-trigger
must-not-trigger
Sink Classification
Ignore
Report Fault
(II) 2nd Level Range Reduction
Binary Decision Diagram-based Unsafe Paths Extraction
(III) Path-sensitive Analysis & Detection

FIGURE 2. RARE Framework.

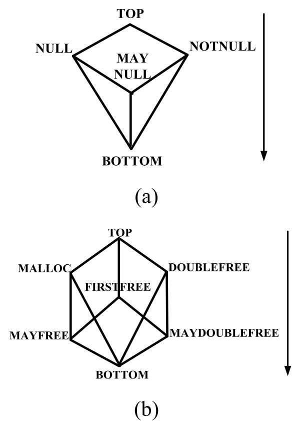

FIGURE 3. Hasse diagrams of NPD and ML fault detection.

null pointer deference (*NPD*) and memory leak (*ML*) faults, the characteristic variables are the pointer variables that are involved in *NPD* and *ML* faults. Whereas for undefined reference (*UDR*) faults, characteristic variables may include all program variables along the execution paths. In our work, we classify def-use faults into the following two categories. Both are handled in the same fault detection framework.

1) EXPLICIT DEF-USE FAULTS

The typical def-use faults in this category include *NPD* and buffer overflow (*BO*) faults. They have very explicit faulty uses. For example, when a pointer dereference (a use site) encounters a *NULL* value that is assigned earlier, a *NPD* fault is triggered. Similarly, when an out-of-range value is assigned to a buffer index variable, a *BO* fault is triggered when it is used later on.

2) NON-EXPLICIT DEF-USE FAULTS

The typical def-use faults in this category include *ML* and *UDR* faults. This kind of faults does not have an explicit def-use pattern or characteristic variables as manifested in the last category. It needs to be cast into a def-use pattern to allow def-use *FA* computation to be carried out in the same fault detection framework. For example, in *ML* faults, memory allocation and de-allocation operations need to be cast as definition sites (i.e. sources), whereas the program exit point is cast as a virtual use site (i.e. virtual sink). When we perform *FA* computation, a *FA* value "*MALLOC-unsafe*" is attached to the pointer variable when a memory allocation function is called. A *FA* value "*FREE-safe*" is attached to the characteristic variable when a free function is called. The only use site that can trigger a *ML* fault is at the program exit point (the virtual use site). If the characteristic variable still carries unsafe *FA* value, such as *MALLOC-unsafe*, a memory leak fault is triggered at the program exit point (i.e. the virtual sink). *UDR* faults are handled similarly. We assign an "*UNDEFINED-unsafe*" *FA* value to each characteristic variable at the program entry point (the virtual definition site). If the characteristic variable still carries an "*UNDEFINED-unsafe*" *FA* value at the program exit point, it will trigger a *UDR* fault.

B. PROPAGATION AND COMPUTATION OF FA VALUES

Performing efficient *FA* value computation and propagation on characteristic variables is critical. In our current framework, we assign different *FA* values to detect different fault types. For example, for *NPD* faults, *FA* values include *NULL*, *NOTNULL* and *MAYNULL*; whereas for *ML* faults, *FA* values include *MALLOC*, *FIRSTFREE*, *DOUBLEFREE*, *MAYFREE*, *MAYDOUBLEFREE*. These *FA* values for different fault types are all propagated together at the same time during the dataflow computation.

Each *FA* value also has a safety attribute, i.e. *safe*, *must-unsafe* and *may-unsafe*as mentioned earlier. The safety attribute is attached to the *FA* value during the dataflow computation. Different safety attributes dictate different actions during fault detection at a use site (see Fig.2).

- ***Safe***: No fault will be reported.
- ***Must-Unsafe***: It marks the use site as a *must-trigge*r sink, and report the fault.
- ***May-Unsafe***: If there are guard statements at the use site, it will change the safety attribute from*may-unsafe* to *safe*, and mark it as a *must-not-trigger* sink. No fault will be reported. For other cases, it is marked as *may-trigger* and requires further determination by the later path-sensitive analysis.

Fig.1 shows an example that involves intra-procedural and inter-procedural fault attribute computation and propagation.

1) INTRA-PROCEDURAL FA VALUE COMPUTATION

Path-insensitive *FA* value computation and propagation is performed during dataflow computation and propagation.

(1) For a branch statement, we perform a meet operation at the branch confluence point, similar to the *phi* operation in the static single assignment (*SSA*) [7]. That is, if the *FA* values from all branches are safe, then the use site is safe, same for the unsafe case. Otherwise, the use site is may-unsafe. For the *may-unsafe* case, we need to determine whether the use site will be executed or not. If it will never be executed, e.g. it has guard statements, it is in the*must-not-trigger* category and no fault will be reported. Otherwise, it is in the *may-trigger* category and requires the path-sensitive analysis later.

(2) For a loop, we perform a limited loop unrolling, i.e. we iterate a few times. It is a quick-and-dirty but effective approach because in most cases, the *FA* values can be determined during the limited loop unrolling, for instance, from *NULL* to *NOTNULL* or *MAYNULL*. The case of transforming from *NOTNULL* or *MAYNULL* to *NULL* is rare.

In Fig.1(b), a branch confluence operation is performed at the definition site of *p* at line 23, which merges *NULL* and *NOTNULL* to a *FA* value of *MAYNULL*. In addition, since there exists a safeguard statement at line 22, even if the definition of *p* is unsafe, it still cannot reach the use site at line 23. We can thus transition its *FA* value from *MAYNULL* to *NOTNULL* at line 23.

2) INTER-PROCEDURAL FA VALUE COMPUTATION

The *FA* computation at a call site is context-sensitive. That is, different call sites may have different fault behavior. The example illustrated in Fig.1(c) shows the difference between the call site at line 18 and that at line 20 of function *boo*(). The result at line 18 is *safe*, and is *may-unsafe* at line 20.

C. FAULT ATTRIBUTE LATTICE

To allow it to fit in the same dataflow infrastructure in a compiler, we transform the fault detection problem to a *FA* computation and propagation problem. To describe the fault attributes and their computation rules, we introduce the Fault Attribute Lattice (*FAL*), which models a *FA* value as a node in *FAL*. The *FA* transition rules are modeled as the operations on *FAL*. A more formal definition of *FAL* and its operations are presented in Section III-A-1.

D. REDUCING PROGRAM RANGE FOR PATH-SENSITIVE ANALYSIS AND FAULT DETECTION

To mitigate the impact of the high cost in more accurate analyses, we use a two-level approach to reduce the program range (i.e. the code regions with the relevant execution paths) using low-overhead analysis approaches.

1) FIRST-LEVEL PROGRAM RANGE REDUCTION

The first-level program range reduction is performed through sink classification during *FA* value computation and propagation. In this phase, whenever a use site acquires a safe *FA* value, it is dropped from further analysis. If it has an unsafe *FA* value, it is further determined whether it is *must-unsafe* or a *may-unsafe*. If it is *must-unsafe*, it is grouped into the *must-trigger* category and reported as a fault. If it is *may-unsafe*, then a later path-sensitive analysis may be required to determine whether it is *safe* or *must-unsafe*.

As mentioned earlier, to identify those use sites with safeguard statements and eliminate them from further path-sensitive analysis, we collect a small set of code templates that have some common safeguarded program patterns. We use a very low cost heuristic pattern matching approach to identify such safe-guarded use sites, and change their safety attribute from *may-unsafe* to *safe*. They are dropped from further analysis. It is noteworthy that although such an engineering approach appears to be ad hoc, it is quite effective in practice in reducing the program range. It is because application programmers usually make them very simple and easy to identify for easy maintenance. As more and more application programmers are using such safeguard statements to improve the robustness of their programs, more such code regions can be dropped for further analysis.

The *may-unsafe* sinks are further classified into either *must-not-trigger* or *may-trigger* category. The group of *may-trigger* sinks and their execution paths form the initial reduced program range for later path-sensitive analysis.

2) SECOND-LEVEL PROGRAM RANGE REDUCTION

To further reduce the program range produced in the first-level, we extract all paths that propagate the unsafe attribute to each *may-trigger* sink because they are the paths that can potentially trigger a fault at the sink. Whether a path propagates a safe or an unsafe attribute to a sink depends on the safety attribute of its last definition along the path. Note that from an unsafe source down to a*may-safe* sink with a *may-trigger*attribute, there could be many paths between them. There may be definitions with the safe attribute that override them, and be propagated to the *may-trigger* sink. It will change the sink's safety attribute to safe, and thus be removed from further path-sensitive analysis.

To facilitate such an analysis, our approach is to encode those paths between an unsafe definition and a *may-trigger*sink with a binary decision diagram (*BDD*) type of encoding scheme. That is, each path will propagate an encoded *BDD* string that records all branch directions (i.e. true branch or false branch) along the path. Source definitions and *may-trigger* sinks also carries such a *BDD* string. The last definition on the path that impacts the safety attribute of the sink must be the definition whose *BDD* string is the longest among all definitions' *BDD* strings reaching this sink (see Section III-C-2 for more details).

III. THE RARE FRAMEWORK

In this section, we describe the formal models, heuristic approaches, and the major components in our framework shown in Fig.2. There are three major components: (1) first-level range reduction, (2) second-level range reduction, and (3) path-sensitive analysis and fault detection.

A. FORMAL MODELS AND APPROACHES

1) FAULT ATTRIBUTE LATTICE (FAL)

A fault attribute lattice (FAL) is an algebra system with three components: V_{FAL}, F$_{intersection}$, and F_{union}, denoted as:

$$FAL = (V_{FAL}, F_{intersection}, F_{union})$$

The three components, V_{FAL}, F$_{intersection}$, and F_{union} are defined as follows.

$$\begin{aligned} V_{FAL} &= \{\perp, \top, falval_1, falval_2 \ldots falval_n\} n \leq 1 \\ F_{intersection} &: V_{FAL} \times V_{FAL} \rightarrow V_{FAL} \\ F_{union} &: V_{FAL} \times V_{FAL} \rightarrow V_{FAL} \end{aligned}$$

V_{FAL} is a set of *FA* values, $falval_i (1 \leq i \leq n)$, that support a particular type of def-use fault detection. For example, for *NPD* faults, they are *NULL*, *NOTNULL* and *MAYNULL* (see Fig.3(a)). $\perp \in V_{FAL}$ represents the initial *FA* values (shown as *TOP*), whereas $\top$ represents the detected *FA* value that will be reported (denoted as *BOTTOM*). *FA* values are carried by characteristic variables and are modeled as lattice values at the *FAL* nodes.

The transition rules of *FA* values are mapped to the lattice operations, which are defined through $F_{intersection}$ or F_{union}. For example, $F_{intersection}$ (*NULL*, *NOTNULL*) = *MAYNULL* in *NPD*(see the edges from *NULL*, *NOTNULL* to *MAYNULL*in Fig.3(a)), or $F_{intersection}$ (*MALLOC*, *FIRST-FREE*) = *MAYFREE* in *ML* (see Fig.3(b)). We perform $F_{intersection}$ at the confluence of branches (see Section III-B-1). In our algorithm, the *FA* values will be iteratively modified until a final *FA* value is reached, which has been proved to always terminate on a lattice with a limited height [8]. A Hasse diagram [9] is a commonly used graphical representation for such lattices. The Hasse diagrams that represent the FALs of the two common *def-use* faults: *NPD* and *ML* are shown in Fig.3.

2) TRIGGER-RELEVANT CONTROL-FLOW GRAPH (TR-CFG)

Given a sink of a characteristic variable v, denoted as $USE(v)$, the fault detection analysis needs the following information: 1) the *definitions* of v, denoted as the set $DEFs(v)$, that can reach the $USE(v)$ through some control-flow paths; 2) $USE(v)$ itself; and 3) the control-flow paths from $DEFs(v)$ to $USE(v)$. A sub-graph of the control-flow graph *(CFG)* that consists of the above three components is called a *Trigger-Relevant CFG (TR-CFG)*. That is, $TR\text{-}CFG_v = < V, E >$, where $DEFs(v) \subseteq V$, and $USE(v) \in V$. $TR\text{-}CFG_v$ is used to analyze the triggering scenarios of v. It is also used in the heuristic *must-not-trigger*pattern matching for the safeguarded sinks and the *BDD*-style path extraction scheme described in Section III-C-2.

3) HEURISTIC CODE PATTERN MATCHING

For *may-unsafe* sinks, we further determine whether they are*must-not-trigger*sinks or not, e.g. with or without safeguard statements. If they are, exclude them from further analysis. Otherwise, group them as *may-trigger* sinks and pass them for second-level range reduction.

The work to determine whether a sink of variable v is a*must-not-trigger* sink is conducted on $TR\text{-}CFG_v$ with heuristic pattern matching. Two of the representative code patterns for safeguarded *must-not-trigger* sinks are shown in Fig.4.

Check Safe Condition:	Example from packet-h245.c (*wireshark*)

if (<*var*>!=<unsafe value>)
{
Use(<*var*>);
}
	if (*h245_pi* !=NULL)
	{
	if (strlen(*h245_pi*->frame_label) == 0)
	{ …… }
	}

Check UnSafe Condition:	**Example from voip_call.c (*wireshark*)**

if (<*var*>==<unsafe value>)
{
Response After Check
}
	if (*pi*->*tap_call_id* ==NULL) {
	return 0; // response after check
	}

	Example from config.c (*wine*)

	if (!*home*)
	fatal_error("could not determine your home directory\n"); // response after check
	
	void fatal_error(const char *err, ...)
	{ …… exit(1); }

FIGURE 4. Code patterns for some safeguard statements.

We search such code patterns for every dominator of a given sink $USE(v)$ on $TR\text{-}CFG_v$. Such code patterns always involve the safeguard statements, which can be quickly identified.

For example, the safeguard statements for *NPD* are often in the form of *"if(x==NULL)"*or *"if(x!=NULL)."* For *ML*, the safeguard statements are in the form of*"if((x=malloc(...))==NULL)."* Whereas the typical actions after such safeguard statements are "*return*" and "*exit*." For *BO* faults, the safeguard statements are often in the form of *"if(x < boundary_val)"* or *"if(x > boundary_val)."* Although such simple pattern matching cannot find all*must-not-trigger*sinks with different patterns of safeguard statements, it is still quite effective in practice. In our experiments, it helps us to identify a large number of*must-not-trigger* sinks, and reduce the range of path-sensitive analysis substantially. More patterns can be included if needed. But we suspect there will be very few, as application programmers usually will not use complicated "tricky" safeguard statements for ease of program maintenance.

B. FIRST-LEVEL RANGE REDUCTION

First-level range reduction includes two parts: (1) aggressive *FA* value computation and propagation, and (2) sink classification, in particular, on *must-not-trigger* sinks.

1) AGGRESSIVE FA VALUE COMPUTATION AND PROPAGATION

A path-*insensitive FA* computation is first performed to classify the use sites into three categories according to their definitions' triggering attributes, i.e. *safe*, *must-unsafe* and *may-unsafe*. It includes inter-procedural context-sensitive *FA* values computation, and intra-procedural, flow-, field-sensitive *FA* values computation.

a: INTER-PROCEDURAL, CONTEXT-SENSITIVE FA VALUE COMPUTATION ALGORITHM

To make the inter-procedural *FA* value computation more accurate, we group the input and output parameters of a function into *Use_Ins* and *Def_Outs*. When we inter-procedurally compute the *FA* values, the information of *FA* values is attached to the *Use_Ins*and *Def_Outs*. We first traverse the call graph in a bottom-up manner. It computes a summary for each function using common intra-procedural summary computation algorithm. Here, the summary records whether the *FA* computation of local variables and*Def_Outs* depends on *Use_Ins* or not.

After the bottom-up computation, we then traverse the call graph in a top-down manner. For each function f visited, we perform the following tasks in a context-sensitive manner. 1) Map *Use_In*'s *FA* value of f from the current caller; 2) instantiate f's summary, and refresh the *FA* values according to the context it is in. Function pointers are taken into consideration by using the result from the pointer analysis when it constructs the call graph. To handle the case of recursive calls, we reduce the call graph to a *Strongly Connected Directed Acyclic Graph* (*SCC-DAG*). It then processes each function by traversing the *SCC-DAG*.

b: INTRA-PROCEDURAL, FLOW-SENSITIVE, AND FIELD-SENSITIVE FA VALUE COMPUTATION ALGORITHM

In intra-procedural algorithm, it visits each statement of a function successively based on the *SSA* representation of the function. We iteratively compute *FA* values for all *FA* related statements based on the *SSA* form until no further *FA* values are changed. For each statement$s$ in function$f$, we perform different computation according to the type of statement s. The compute rules are as follows:

$$\Delta 1(s, assign, \iota_s) ::= \digamma 1(FA(\Upsilon_s)) \tag{1}$$

$$\Delta 2(s, call, o\rho_s) ::= \digamma 1(FA(\tau\rho_s), \eta_{\text{summary}}) \tag{2}$$

Here, $\Delta(x, y, z)$ denotes a computation of characteristic variable z's *FA* value from statement x whose statement type is y, e.g. an assignment or a call, as shown above in (1) and (2), respectively.

"$m ::= n$" means m depends on n. $\digamma 1(p_1, p_2, \ldots p_n)$ is a computation function that takes $p_1, p_2, \ldots p_n$ as arguments. Formula (1) indicates that if s is an *assignment* statement, the *FA* value of its ι_s, i.e. the left-hand side, is computed according to its right-hand side expression, which is $FA(\Upsilon_s)$. $\digamma 1$ describes such a dependent relation. Note that if the expression involves characteristic variables, the $F_{intersection}$ function of the lattice is used. Formula (2) indicates that if s is a *call* statement, the *FA* values of the output parameters $o\rho_s$ are computed by applying the *FA* values of input parameters $FA(\tau\rho_s)$ to the function's summary η_{summary}.

In addition, we perform a $F_{intersection}$ operation at a branch's confluence point. For those statements that involve pointer operations, the point-to information is used to compute the *FA* values conservatively. We try to use a pointer analysis that is scalable and accurate. It helps us achieve overall scalability and accuracy.

The integrated framework of inter- and intra-procedural algorithm is illustrated in Algorithm 1. Where *FA*(t,$s1$,$s2$, ...sm) computes t's *FA* value using $s1$, $s2$,...sm, m≥1. The intra-procedural algorithm *FA_Val_CP* has a polynomial time complexity. This is because the intra-procedural computation is *monotonously upward* [8], and the iteration can be terminated within the height of *FAL*. For the inter-procedural algorithm, the worst case is when each statement is a call statement, and summary re-computation is needed on every call site. The time complexity can be estimated to be $O(h * n * n)$. It is close to $O(n^2)$ as h is the height of *FAL*, which is a small constant, and n is the number of program statements.

2) SINK CLASSIFICATION

Sink classification is performed along with *FA* value computation and propagation, which also takes the safety attributes into consideration as shown in Algorithm 2.

For those *must-unsafe* sinks, we can also characterize them as *must-trigger*sinks and report the faults (lines 3-5). For those *may-unsafe* sinks, the heuristic patterns matching

Algorithm 1 FA Values Computation and Propagation

```
1   Procedure FA Val CP (CallGraphG)
    /*START: Inter_Compute (CallGraphG); */
2   BuildG with pointer analysis information;
3   Find Strongly Connected Component ( SCC) of G;
4   Build SCC-DAG of G ;
5   For each node f of SCC-DAG in bottom-up order
6       Intra_Summary(f);
7   End For
8   For each node f of SCC-DAG in top-down order
9   For each incoming edge of f in G
10      Map Use_In'sFA value of f from caller;
            /* START:Intra_Compute(f); */
11  While ( theFA value changed )
12          For eachsinf
13  If (s is an assignment) then
14              Compute FA(LHS(s),RHS(s));
15  End If
16          If (sis a call) Then
17  Compute FA(Def-Outs, Use-In,summary);
18              End If
19              If (sis a branch convergence) Then
20       Compute   the FA(LHS(s),$F_{\text{intersection}}$ );
21      End If
22      End For
23  End While
      /* END:Intra_Compute(f); */
24      End For
25    End for
26      ComputeFAvalue for each node in each SCC;
          /* END: Inter_Compute(CallGraphG); */
27  End
```

Algorithm 2 First-Level Range Reduction

```
1   Procedure Sink_Classification ( f )
2     For each s in f
      /* Path-insensitive analysis */
3     If (ause-site of s carries unsafe FA value) Then
4     If (theFA value is with Must attribute) Then
      /* must-trigger sinks */
5       Report the fault;
6     Else
      /* Limited path conditions analysis */
7     For each dominator of s
8   If (pattern is matched) Then
      /* must-not-trigger sinks */
9       Ignore the current use site;
10      End If
11    End For
12  If (no pattern matched in all dominators) Then
    /* may-trigger sinks */
13  Put the use-site and its definition into work list for
    second-level path-sensitive analysis range
      reduction;
14          End If
15          End If
16        End If
17    End For
18  End
```

algorithm is applied (lines 6-11). The general strategy to identify *must-not-trigger* sinks is to check every dominator of a given sink towards its corresponding source in *TR-CFG*. The check only needs to be on the fault-related statements.

For example, for a sink that hold the safeguard statement such as "if (< variable >==< unsafe value>)" (see "*check unsafe condition"* in Fig.4), we will search whether there is a response action such as *exit*or*function return* or not. If the pattern matches, the current sink will be excluded from further consideration. Otherwise, it will be marked as a *may-trigger* sink for further analysis. Another example is for sinks that have safeguard statement such as "if (<variable>!=<unsafe value>)" (see "*check safe condition"*and the response after the check in Fig.4), we search their dominators towards their corresponding sources in *TR-CFG* to see if such pattern matches. If we find such a match, they can also be excluded from further analysis.

The remaining sinks are automatically grouped into*may-trigger* sinks category (lines 12-14), and placed on the work list for second-level range reduction. The time complexity of algorithm 2 can be estimated to be O($ns*ds$), where $ns$and$ds$are the number of sinks and dominators, respectively.

C. SECOND-LEVEL RANGE REDUCTION

After the first-level range reduction, only the *may-unsafe* sinks with the *may-trigger* attribute remain to be further analyzed at the second-level. A naive approach will be to use program slicing techniques [10] to extract the statement set that is relevant to each *may-trigger* sink and its corresponding source, and apply path-sensitive analysis on them.

One common technique is to perform a backward slicing on a *may-trigger* sink first and get the statement set S_{BT}; followed by a forward slicing from an identified unsafe source to get the statement set S_{FS} (assume the unsafe source is reachable). Then find the statement set S_{TS}, which is the intersection of the sets S_{BT} and S_{FS}. S_{TS} includes the statements on the execution paths between the source and the sink. The main drawback of such a slicing approach is that the collected execution paths not only include the *unsafe* paths but may also include many *safe* paths, which are not required to be further analyzed. To avoid such a problem, we use an extraction scheme that only extracts *unsafe* paths for the *may-trigger*sinks identified in the first-level range reduction. It is based on *Binary Decision Diagram* (*BDD*) path encoding scheme.

TABLE 1. BDD encoding rules.

Rules	Representation
Atom	*Bdd (branch, true) → 1* *Bdd(branch, false) → 0*
Multiplication	*∏ (Bdd(bi,vi)) → PBdd(path)* *∏ (Bdd(bi,vi))=Bdd(b1,v1)·* *Bdd(b2,v2)·…·Bdd(bn,vn) 1≤i≤n*
Confluence	*Conf (PBdd(p1),PBdd(p2))* *→PBdd(p1) || PBdd(p2)*

1) PATH CONDITION ENCODING USING BDD

We encode all branch outcomes (*true* or *false*) along an execution path in the style of *BDD*. The main encoding rules are listed in Table 1. They consist of *atom rules*, *multiplication rules* and *confluence rules* specifically for our encoding purposes. More complex rules can be generated by a combination of the above basic rules. The result of a *BDD*encoding is called a *BDD item*, also denoted as *BDD* if there is no confusion.

In Table 1, *Bdd*(*b*,*v*) denotes the *BDD* item when branch *b* takes the value of *v*,which can be either*true*or*false*. For example, *Bdd*(*branch*, *true*) → 1 indicates using *BDD* item 1 to represent a *true* branch; *PBdd*(*path*) denotes the *BDD* item of a *path*, which can be a string of 0 and 1 that includes all the branch outcomes along the path; *Conf*(*x1*,*x2*) denotes the *BDD* item that is a *concatenation* of *BDD* items*x1*and *x2* at the confluence of a branch. *BDD* items are attached to all sources, sinks, and branch conditions.

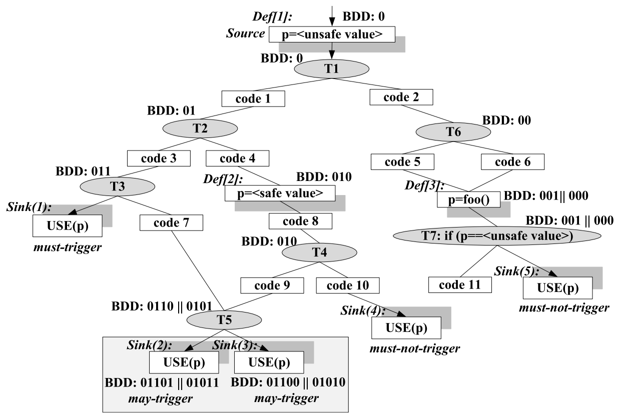

FIGURE 5. An BDD encoding example.

In the example shown in Fig.5, the source is attached with *BDD* "0"; conditions T2, T5 are attached with *BDD* "01" and *BDD* "0110 || 0101," respectively; Sink(2)'s *BDD* is "01101 || 01011."

2) PREFIX-MATCHED UNSAFE PATHS EXTRACTION

By performing *a prefix analysis* on a sink's *BDD* items, we can determine the order of the definitions on the path to the sink. This is because the *last* definition that assigns the value to the sink must have the longest *BDD* item among all other definitions along the same path. By combining all *BDD* items' safety attributes; the safety attribute of the sink can be determined.

For each *may-trigger* sink, we perform the following steps to extract *unsafe* paths (shown in Algorithm 3, whose time complexity is affected by the number of edges in the CFG).

Algorithm 3 Second-Level Range Reduction

```
1   Procedure Path_Encoding_and_Extraction
    (P_unsafe)
    /* Process each may-trigger sink s   */
2   For each s in Trigger-Unknown sink set
    /* Compute s'W_def */
3   Collectdefinitions of s to S_def inter-procedurally
4   Records's related function to S_func
5               For eachf in S_func
6       While not finished traversing onf's CFG
7           Encode and record each BB 's BDD  item
8       For each d located in f
9   Encode and record d 's BDD item
10  End For
11      End While
12      End For
13          Descend d 's BDD item s and formW_def
                    /* Collect s'  S_sink_bdd */
14  Collects 's BDD   items to form S_sink_bdd
    /* Peform prefix matching */
15              For each dbdd in  S_def_bdd
16          For each sbdd in  S_sink_bdd
17              If dbdd is a prefix ofsbdd Then
18              If dbdd is unsafe Then
    /* Get an unsafe path  */
19                  Put sbdd into P_unsafe
20                      Delete current sbdd from  S_sink_bdd
21                     Else
                       /* Exclude a safe path */
22                      Delete currentsbddfrom S_sink_bdd
23                 End If
24              End If
25           End For
26      End For
27      End For
28  End
```

Step 1: Find all its sources inter-procedurally, perform *BDD* encoding related to each source, and attach the *BDD* items to its basic block.

Step 2: Sort all sources' *BDD* items in a descending order according to their lengths. Arrange them to form a worklist W_{defs} for the *may-trigger* sink (lines 13).

Step 3: Collect all *BDD* items of this *may-trigger* sink to form set S_{sink_bdd} (lines 14), then decide whether the corresponding path of each element in the set is *safe*or *unsafe*. Check the element s_{curr} in S_{sink_bdd} one by one (lines 15-26):

Step 4: For an *unsafe* definition def_{unsafe} in W_{defs}, if s_{curr} takes def_{unsafe}'s *BDD* as a prefix, then s_{curr} is removed from

S_{sink_bdd} and added it to the unsafe path set P_{unsafe}. Otherwise, select next item in S_{sink_bdd} as s_{curr}.

Step 5: For a *safe* definition def_{safe} in W_{defs}, if s_{curr} takes def_{unsafe}'s *BDD* as a prefix then exclude it from S_{sink_bdd}.

Repeat **Step 4** and **Step 5** until S_{sink_bdd} becomes an empty set Φ, or all items in W_{defs} have been processed.

For example, in Fig.5, the final unsafe paths set P_{unsafe} is {"01101," "01100"}. There are 6 paths excluded from all 8 paths. It is a considerable reduction in the number of paths.

D. PATH-SENSITIVE ANALYSIS AND DETECTION

In our RARE framework, a potential fault is reported when (1) a *must-trigger unsafe*sink is identified in the two-level range reduction phase, and (2) a*may-trigger* sink is confirmed to be *unsafe* along some feasible execution paths, i.e. it is confirmed to be a *must-trigger* after the path-sensitive analysis.

As the program range has been reduced substantially in the two-level range-reduction phase already (see Fig.7), in our current implementation, we use an existing path-sensitive analysis tool (such as Clang-SA) with the guide of our *BDD* information to check if an *unsafe* path is feasible for a *may-unsafe* sink or not. If it is actually a *must-trigger unsafe* sink, then report such a sink as a potential fault.

IV. EXPERIMENTAL RESULTS

A. PROTOTYPE AND BENCHMARKS

We implement our prototype in Open64 [6], an open-source compiler with a very sophisticated analysis tool set. We also implement our path-sensitive analysis for the range-reduced *may-trigger* sinks using the enhanced *BDD* information as described in Section III in Clang-SA [1]. We also use more accurate and scalable pointer analysis LevPA [11] in our implementation. Our experimental platform is an AMD Opteron hex-core CPU server, with a clock rate of 2.11GHz, and a memory size of 48GB. All experiments are done in a single process environment to avoid interferences.

For benchmarks, we selected five open-source realistic applications: *openssh* [12], *sendmail* [13], *httpd* [14], *wine* [15] and *wireshark* [16]. The benchmark set covers a wide spectrum of application areas such as network-level security enhancement tool, internet email routing facility, operating system compatibility-layer application and network protocol analysis. The total code size exceeds 4.8 million lines. The largest individual program *wireshark* has more than 2 million lines.

B. RESULTS AND ANALYSIS

1) FIRST-LEVEL RANGE REDUCTION: SINK CLASSIFICATION RESULTS

One unique feature of RARE is its classification of potential fault statements into different types, and the use of different fault detection schemes with varying degrees of overheads tailored to each type. For *must-trigger*and *must-not-trigger* sinks, we use a low overhead approach. Only for *may-trigger* sinks, we use a relative expensive path-sensitive approach. Consequently, the effectiveness of this approach is to a large extent dependent on the percentage of these three sink categories in a program.

Furthermore, note that the *FA* values of all def-use fault types (i.e. *NPD*, *BO*, *ML*, *UDR* faults) are computed and propagated at the same time during the fault analysis. However, *ML* and *UDR* are of non-explicit def-use types (see Section II-A), and only *NPD* and *BO* have explicit sources and sinks. Between the two, *NPD* has the most complicated and significant triggering characteristics. Thus, *NPD* fault detection constitutes the major portion of the overall def-use fault analysis time. We thus collect statistics based on *NPD* fault detection in our measurements.

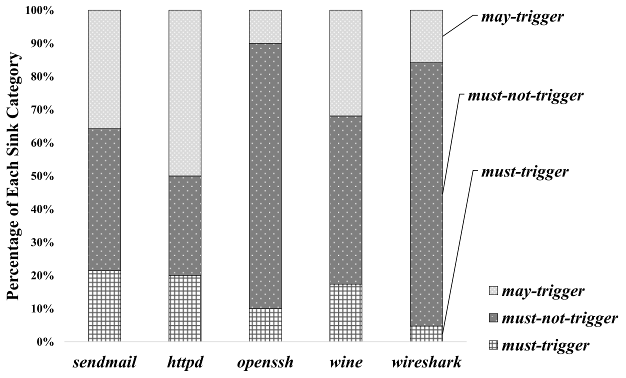

FIGURE 6. The percentage of each sink category in each benchmark for NPD detection.

The percentages of three different sink categories for *NPD* faults in each benchmark are shown in Fig.6. For most of our benchmark programs, the combined percentage of *must-trigger* and *must-not-trigger sinks*in each individual program exceeds 84%. Especially in *openssh* and *wireshark*, the percentage of *must-not-trigger sinks* alone is greater than 79%.

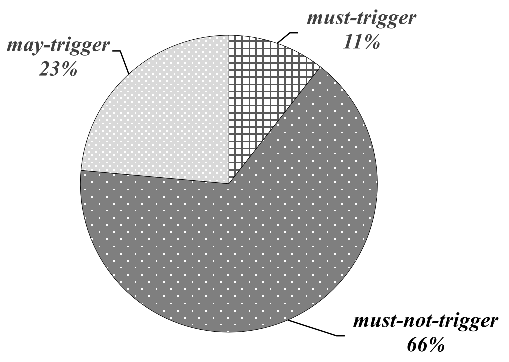

FIGURE 7. The percentages of three sink categories in all benchmarks for NPD detection.

Fig.7 shows the percentage of three different sink categories in *all* of the benchmarks combined. It shows that only 23% of sinks are *may-trigger,* and the remainder is either *must-trigger* or *must-not-trigger*. These results show that we can *substantially* reduce the program range that requires expensive path-sensitive analysis to improve scalability.

Note that the number of the *must-not-trigger* sinks in *wiresharkis* very large (see Fig.6). It distorts the distribution in Fig.7.

As mentioned before, the safeguard statements are very common and often required in large application in practice. It is particularly important to take such program characteristic into consideration when we try to reduce the number of *may-trigger* sinks and avoid too many false positives.

TABLE 2. Number of safeguard statements in all benchmarks.

Programs	Size (KLOC)	NPD #STMT$_{sg}$	BO #STMT$_{sg}$
openssh-6.7p1	85.5	≥1742	≥1
sendmail-8.15.1	87.4	≥2304	≥0
httpd-2.2.27	231.3	≥211	≥2
wine-1.1.27	1734.8	≥3966	≥578
wireshark-1.12.4	2688.1	≥4807	≥461

Table 2 shows the number of safeguard statements in each benchmark program in the 3rd and 4th column under #STMT$_{sg}$ for *NPD* and *BO* faults, respectively. The 2nd column shows the size of each program in *thousand lines of code* (KLOC). Except in *httpd*, there are more than thousands of *NPD* safeguard statements for *must-not-trigger* sinks in all benchmarks for *NPD* faults.

2) SECOND-LEVEL RANGE REDUCTION: REDUCTION RATIO COMPARED TO SLICING-BASED APPROACHES

To compare the effectiveness of using a BDD-based scheme in the second level of our program range reduction, we use a program-slicing scheme based on [17]. It is very similar to the program slicing approach described in Section III-C. The results are shown in Table 3. The average reduction ratios on program size using RARE and slicing-based strategy are 66% and 48%, respectively, i.e. we reduced additional 18% of program range on average compared to the program slicing-based approach.

3) DETECTION TIME AND MEMORY USAGE

The detection time of RARE consists of three major parts: (1) the time for the first-level range reduction; (2) the time for the second-level range reduction; and (3) the time for the path-sensitive analysis and fault detection after the two-level range reduction.

The results are shown in the 5th column and 9th columns (marked as *1st*) in Table 4 for the first part (i.e. the first-level range reduction). It includes the time to detect the *must-trigger* sinks and to exclude the *must-not-trigger* sinks. The 6th and 10th (marked as *2nd*) columns show the time for the second part (i.e. the second-level range reduction). It includes the time to extract unsafe path set (using a BDD-based approach) for each*may-trigger*sink. The 7th and 11th columns (marked as *PS)* show the third part for the time that uses a path-sensitive analysis to determine the feasibility of the extracted unsafe paths and check whether a *may-trigger* sink is a fault or not. It is worth noting that the time for path feasibility determination will be greatly affected by the ability of third party tool, this is also the part for further improvement.

We compare our approach with Clang-SA, which uses a path-sensitive analysis on all execution paths and is available in popular Clang compiler. As shown in Table 4, the analysis time in the first two parts, i.e. the two-level range reduction, is much less than that of using Clang-SA. Clang-SA requires exploring *path feasible states*. But it processes only simple cases. For example, Clang-SA ignores the condition expressions that consist of more than one variable. Even so, the required exploration space is still quite considerable for large application programs.

Currently, to simplify our implementation, we use Clang-SA in the third part to determine the feasibility of the extracted unsafe paths, and check whether a *may-trigger* sink is a fault or not. Its time is shown in the columns under *PS*. It shows significant reduction in time compared to Clang-SA due to the reduced range. However, the total fault detection time of RARE shown in the 8th and 12th columns under *Total* is impacted somewhat by the implementation of Clang-SA. Nevertheless, the total detection time shown in the last row of Table 4 still shows (5294-4362)/5294 = 17.6% improvement for *NPD*, and (5879-4755)/5879 = 23.6% improvement for *ML* over Clang-SA.

The memory usage by RARE is mostly determined by the program analysis, in particular the pointer analysis, and the two-level range reduction. It varies in different phases. In general, pointer analysis is the most memory consuming because it needs to keep a large amount of pointer analysis information. Whereas the space overhead of *FA* value computation and propagation is relatively small because it only keeps the fault attribute values. Our *BDD* encoding and unsafe path extraction also uses less memory space compared to that used in program slicing approaches. It generally uses less memory space than that used in pointer analysis. Taking the largest benchmark *wireshark* as an example, the average memory usage is only about 2300 MB.

4) DETECTION ACCURACY

Table 5 shows the *NPD* and *ML* detection results by Clang-SA and RARE. In Table 5, *#FR* denotes the total number of faults reported. It includes both the *true*faults and the *false positives*. *#FP* denotes the number of false positives after we *manually* check each of the reported faults. *#RFN* denotes the number of false negatives. As there is no *golden* standard for the benchmarks we used, we collect all of the true faults (verified manually) reported by both RARE and Clang-SA. The number of false negatives *#RFN* is obtained by comparing the detected true faults in each approach against the combined set of all true faults**.**

For Clang-SA, the loss of detection accuracy is mainly caused by two factors. One is that it does not check safe-guarded statements for *must-not-trigger*sinks. This results in many false positives. For example, many false positives in *wireshark*, *httpd*, and *sendmail* are in this category. It is also

TABLE 3. Program size reduction.

Program	Program Size				
	Original Size (KLOC)	Slicing-Based Strategy		RARE	
		Range Reduction (KLOC)	Reduction Ratio	Range Reduction (KLOC)	Reduction Ratio
openssh-6.7p1	*85.5*	53.2	62%	65.19	76%
sendmail-8.15.1	*87.4*	39.3	45%	54.13	62%
httpd-2.2.27	*231.3*	44.7	19%	109.61	47%
wine-1.1.27	*1734.8*	732.2	42%	1133.05	65%
wireshark-1.12.4	*2688.1*	1885.7	70%	2164.98	81%
Avg.		*551.02*	*48%*	*705.39*	*66%*

TABLE 4. Detection time(in seconds).

Program	Original Size (KLOC)	Clang-SA		RARE							
		NPD	ML	NPD				ML			
				1st	2nd	PS	Total	1st	2nd	PS	Total
openssh-6.7p1	*85.5*	126	280	26	38	51	115	12	37	23	72
sendmail-8.15.1	*87.4*	119	265	18	29	33	80	20	45	47	112
httpd-2.2.27	*231.3*	889	920	11	103	118	232	3	40	65	108
wine-1.1.27	*1734.8*	378	398	72	192	138	402	72	268	202	542
wireshark-1.12.4	*2688.1*	3782	4016	901	1356	1276	3533	730	1998	1193	3921
Tot.	*4827.1*	*5294*	*5879*	*1028*	*1718*	*1616*	*4362*	*837*	*2388*	*1530*	*4755*

TABLE 5. NPD and ML faults detected.

Program	NPD						ML					
	Clang-SA			RARE			Clang-SA			RARE		
	#FR	#FP	#RFN	#FR	#FP	#RFN	#FR	#FP	#RFN	#FR	#FP	#RFN
openssh-6.7p1	26	20	0	9	3	0	2	2	0	0	0	0
sendmail-8.15.1	10	8	0	2	0	0	0	0	4	5	1	0
httpd-2.2.27	12	5	0	8	1	0	0	0	0	0	0	0
wine-1.1.27	14	5	0	10	1	0	6	1	70	84	9	0
wireshark-1.12.4	17	5	0	14	2	0	0	0	33	36	3	0
Avg. R_{FP}	*54%*			*16%*			*38%*			*10%*		
Avg. R_{RFN}	*0%*			*0%*			*96%*			*0%*		

true for many false positives *ML* faults in *openssh*. Another reason is that it often treats infeasible paths as feasible because it can only handle limited forms of branch conditions. In comparison, RARE reduced about 66% of false positives. It plays an important role in improved detection accuracy.

In Table 5, we also show the *false positive* ratio R_{FP}= *(#FR - #DTF) / #FR*, in which *#DTF* is the number of detected true faults in each approach. It shows the percentage of true faults detected in each approach. We also show the *false negative* ratio $R_{RFN} = (\#TF - \#DTF)/\#TF$, in which *#TF* is the total number of true faults detected from both RARE and Clang-SA as explained earlier.

We found that the R_{FP} of Clang-SA for *NPD* and *ML* are about 54% and 38%, respectively, which are quite high. This shows that our approach of combining two-level range reduction, BDD-based path extraction scheme and checking of safeguard statements are quite effective. Furthermore, the average R_{RFN} for *ML* faults in Clang-SA reaches to a high of 96%. It is distorted by the results from *wine* and *wireshark*. It is mostly due to poor inter-procedural analysis and wrapper function identification.

In summary, the average false positive ratio of RARE is 16% (*NPD*) and 10% (*ML*), respectively, which is about a third of Clang-SA's. There is also no false negative on the known faults by RARE. We found that RARE's high detection accuracy is also due to the following two factors. 1) It eliminates many false positives through the identification of *must-not-trigger* patterns (many from identifying safeguard statements). 2) It uses highly accurate program analysis such as the inter-procedural pointer analysis done by LevPA [11].

V. RELATED WORK

PATH-INSENSITIVE DETECTION

Splint [18] is a path-insensitive annotation-assisted static detection tool that performs detection very quickly but with lots of false positives. FindBugs [19] is also a representative path-insensitive detection tool that employed dataflow analysis in bug finding in Java code. It suffers the accuracy problems like other path-insensitive detection approaches.

PATH-SENSITIVE DETECTION

Saturn [2], [20] is an accurate path-sensitive analysis and detection framework. It transforms the program constructs into boolean constraints, and use a boolean satisfiability (SAT) solver to infer and check faults. But since it exhaustively checks all execution paths one by one, the overhead is very large for large programs. Clang-SA [1] is also a path-sensitive detection tool. To handle the scalability problem, it only performs *intra*-procedural path-sensitive analysis and restricts the types of path condition expressions it can process to reduce the overhead. The restrictions often resulted in a big loss in detection accuracy. Marple [4] employs a demand-driven path-sensitive approach for scalability. It first finds all potentially faulty statements in a program, and then examines paths from each potentially faulty statement to the program entry and see whether a fault occurs or not. This is supported by some security policies and queries about faults. Marple only excludes the fault-unrelated paths. There could still be a large number of fault-related paths left unanalyzed.

FLOW-BASED DETECTION

Parfait [21] is a static layered program analysis detection framework developed by Sun Microsystems. It uses a simple program analysis technique with low overhead to analyze easy to detect faults, and only does expensive program analysis on complicated faults. The program analysis involved in Parfait includes constant propagation, partial evaluation and symbolic analysis etc. SABER [22] performs fault detection based on a full-sparse value-flow analysis. It reasons about the path conditions guarding the flow of a value only on the relevant parts instead of the entire CFG. But it requires building sparse value-flow graph. It is primarily for memory leak faults.

VI. CONCLUSION

In this paper, we propose a range-reduced two-level def-use fault detection framework, which takes fault triggering characteristics into consideration, leveraging path-insensitive and path-sensitive analysis on different fault triggering scenarios. Using such an approach both accuracy and scalability for large application programs can be successfully addressed. Such an approach can also fit quite well in the dataflow framework in exiting compilers. Our experimental results on more than 4.8 million lines of code (MLOC) in some large applications in practice demonstrate that such an approach is quite scalable and accurate.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their valuable comments.

REFERENCES

[1] *Clang Static Analyzer*. Accessed: Dec. 2017. [Online]. Available: http://clang-analyzer.llvm.org/

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the saturn project," in *Proc. PASTE*, San Diego, CA, USA, 2007, pp. 43–48.

[3] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. STOC*, Shaker Heights, OH, USA, 1971, pp. 151–158.

[4] W. Le and M. L. Soffa, "Marple: Detecting faults in path segments using automatically generated analyses," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 18:1–18:38, Jul. 2013.

[5] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516, Jun. 1978, doi: 10.1109/TC.1978.1675141.

[6] *Open64*. Accessed: Dec. 2017. [Online]. Available: http://www.open64.net/

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Programm. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, doi: 10.1145/115372.115320.

[8] A. V. Aho, R. Sethi, and J. D. Ullman, "Foundations of data-flow analysis," in *Compilers: Principles, Techniques and Tools*, 2nd ed. Hoboken, NJ, USA: Pearson Education, 2006, ch. 9, sec. 3, pp. 618–623.

[9] B. Garrett, "Types of Lattices," in *Lattice Theory*, 2nd ed. Providence, RI, USA: AMS, 1948, ch. 1, sec. 3, pp. 4–6.

[10] M. Weiser, "Program slicing," in *Proc. ICSE*, San Diego, CA, USA, 1981, pp. 439–449.

[11] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang, "Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code," in *Proc. CGO*, Toronto, ON, Canada, 2010, pp. 218–229.

[12] *OpenSSH*. Accessed: Dec. 2017. [Online]. Available: http://www.openssh.org/

[13] *Sendmail*. Accessed: Dec. 2017. [Online]. Available: http://www.sendmail.com/

[14] *Apache*. Accessed: Dec. 2017. [Online]. Available: http://www.apache.org/

[15] *Wine*. Accessed: Dec. 2017. [Online]. Available: http://www.winehq.org/

[16] *Wireshark*. Accessed: Dec. 2017. [Online]. Available: http://www.wireshark.org/

[17] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 12:1–12:41, 2012, doi: 10.1145/2187671.2187674.

[18] D. Evans, "Static detection of dynamic memory errors," in *Proc. PLDI*, Philadelphia, PA, USA, 1996, pp. 44–53.

[19] *FindBugs*. Accessed: Dec. 2017. [Online]. Available: http://findbugs.sourceforge.net/

[20] I. Dillig, T. Dilig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *Proc. PLDI*, Tucson, AZ, USA, 2008, pp. 270–280.

[21] C. Cifuentes and B. Scholz, "Parfait: Designing a scalable bug checker," in *Proc. SAW*, Tucson, AZ, USA, 2008, pp. 4–11.

[22] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Trans. Softw. Eng.*, vol. 40, no. 2, pp. 107–122, Feb. 2014, doi: 10.1109/TSE.2014.2302311.

LUJIE ZHONG received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2013. She joined Capital Normal University, Beijing, in 2002, where she is currently an Associate Professor with the Information Engineering College. She has authored over ten technical papers in prestigious international journals and conferences. Her research interests include software reliability, program analysis, and advance compiling technology.

PEN-CHUNG YEW (F'98) is currently a Professor with the Department of Computer Science and Engineering, University of Minnesota at Twin Cities, Minneapolis, MN, USA. His current research interests include system virtualization, compilers, and architectural issues related to multi-core/many-core systems.

WEI HUO received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2010. He is currently an Associate Professor with the Institute of Information Technology, Chinese Academy of Sciences. His research interests include program analysis and bug detection.

XIAOBING FENG received the Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences, in 1999. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include program optimizing, high performance computing, and program analysis.

FENG LI received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2013. She is currently an Associate Professor with the Institute of Information Technology, Chinese Academy of Sciences. Her research interests include program analysis and fault localization.

ZHAOQING ZHANG is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Her research interests include computer architecture and advance compiling technology.