# Cause Points Analysis for Effective Handling of Alarms

Tukaram Muske
TRDDC, Tata Consultancy Services, Pune, India.
Email: t.muske@tcs.com

Uday P. Khedker
Indian Institute of Technology, Bombay, India.
Email: uday@cse.iitb.ac.in

***Abstract*—Static analysis tools are widely used in practice to improve the quality and reliability of software through early detection of defects. However, the number of alarms generated is a major concern because of the cost incurred in their manual inspection required to partition them into true errors and false positives. In this paper, we propose a static analysis to identify the causes of alarms generated by a client static analysis. This simplifies the manual inspections and reduces the cost involved. The proposed analysis involves the following: (1) modeling the basic reasons for alarms as alarm cause points of several types, (2) ranking these cause points based on three different metrics, (3) a workflow in which a user answers queries about the cause points and the answers are used in subsequent round of the client analysis. The collaboration between the user and the client analysis helps the tool to resolve the unknowns encountered during the analysis and weeding out the alarms. It also helps the user expedite the manual inspections of alarms. Further, the ranking of cause points helps to prioritize the alarms. Our experimental evaluation in several settings demonstrated that the proposed approach (a) reduces manual effort by 23% to 72% depending on various parameters, with an average reduction of 42%, and (b) is also effective in identifying the alarms that are more likely to be true errors.**

## I. Introduction

Static analysis tools have proved their effectiveness in improving the quality and reliability of software through early detection of defects [1]. However, given that verification problems are undecidable in general, reporting of alarms by these tools is inevitable [2]–[4]. Further, the tools compromise on precision to achieve analysis scalability, soundness, or improved performance [5]–[7]. As a consequence, the number of alarms reported is large and a significant amount of effort is spent in their manual inspection required to classify them into true errors and false alarms [3], [7]–[9]. In spite of the cost, the manual inspections are essential to fulfill the practical needs of constructing trusted (safety critical) software [10], [11].

Several studies [12]–[14] report that the manual effort spent in inspecting the alarms is the foremost reason for underuse of (sound) static analysis tools. The effort in manual inspection can be reduced either by increasing the analysis precision or by simplifying the inspection of alarms. The former has received significant attention but is limited by the inherent limitations of static analysis, while the latter has gained attention recently [2]–[4], [15]–[19]. Any approach that addresses the latter ought to be human-centric and must deal with the methodology and tool support being used during the inspections [12], [13].

In this paper, we propose a static analysis for discovering the causes of alarms generated by a client static analysis (eg. an analysis to discover possible array out of bound accesses). Our primary goal is to simplify the manual inspections of alarms and reduce the human effort involved. A secondary goal is to inspect (resp. uncover) as many alarms (resp. true errors) as possible in a given inspection time. The following aspects make the proposed analysis novel.

1) The statements in which the unknown values relevant to the analysis originate, are modeled as alarm cause points by performing a cause points analysis. Instead of reporting alarms generated by a client analysis, we report the cause points of the alarms. This shifts the focus of manual inspections from alarms to their cause points thereby simplifying the process considerably.
2) The alarm cause points are categorized into different types and are ranked using a pragmatic approach based on three metrics: cause point type, contribution score, and their similarity or spatial proximity. The ranking enables inspecting (resp. uncovering) as many as alarms (resp. true errors) possible in a given inspection time.
3) Alarm cause points-specific queries are formed seeking abstract information that relates to the safety at the alarm points (similar to [2], [16]). Answering such queries is much easier and faster than inspecting the alarms.
4) An iterative workflow is used in which the client analysis and seeking user inputs at the alarm cause points are interleaved. The user inputs are received as answers to the queries formed for cause points of alarms generated by the client analysis. Subsequent client analysis is guided by the user inputs. The process is repeated until no more alarms are generated by the client analysis. This approach enables a much quicker convergence on alarms handling.

Our experimental evaluation in several settings demonstrated that the proposed approach reduces the manual effort by 23% to 72% depending on various parameters in the static analysis and manual inspections. On an average, the reduction was by about 42%. Further, the experiments demonstrated that the approach is also effective in identifying alarms that are more likely to be true errors.

The key contributions of this paper are: (1) Cause points analysis for systematic modeling of cause points of alarms and shifting the focus in manual inspections from the alarms

```
1  const int arr1[]={0,3,7,9,14,22,34};
2  char arr2[35], str[20], bound, tmp;
3
4  void foo(){
5    unsigned int i, j, k, length;
6    ... // some code
7    scanf("%s",str); //Cause point CP7
8    if(i < 7 && j < i)
9      bound=arr1[i]-arr1[j];//Cause point CP9
10
11   for(k = 0; k <= bound; k++){ //OFUF
12     f1(k);
13   }
14   length = strlen(str);
15   f2(bound, length);
16 }
17
18 void f1(int p){
19   if(nondet())  arr2[p] = 0; //AIOB
20   else  arr2[p] = 1; //AIOB
21 }
22
23 void f2(int p, unsigned int q){
24   arr2[p - 1] = 100 / q; //AIOB, ZD
25   tmp = str[q]; //AIOB
26 }
```

Fig. 1. Examples of static analysis alarms along with their cause points

to their cause points. (2) A pragmatic ranking scheme using a mix of three metrics to prioritize the cause points. (3) A novel approach of presenting the results of a client analysis to the user by interleaved rounds of the client analysis and seeking user inputs at cause points. (4) Study of alarm cause points and evaluation of our approach in practice.

Our analysis technique is applicable to both sound and unsound static analysis tools alike. However, we limit the scope of the client analyses considered in the paper to sound static analysis tools that perform analysis based on the data-flow information and to the domains where resolving of all the reported alarms is warranted (like safety critical systems). The checking of user provided assertions, and alarms reported based on structural information or local bug patterns (like FindBugs [20]) are excluded from the scope of client analyses. *Paper outline.* Section II provides a motivating example. Section III presents the modeling of alarm cause points and their ranking is presented in Section IV. Section V describes framing of the cause points-specific queries and a framework for the effective user interactions with an analysis tool. Section VI presents our experimental evaluation. Related work is presented in Section VII. Section VIII concludes the paper.

## II. Motivating Example

Consider the example in Figure 1 adapted from a real-life embedded system; simplified considerably for exposition, yet sufficiently rich to present our ideas. A sound static analyzer (like TCS ECA [21]) generates six alarms[1] for this example: four instances of array index out of bound (AIOB), and one instance of each division by zero (ZD) and overflow underflow (OFUF). The analyzer reports these alarms because it is unable to discover the precise values of variables *str* at line 7 (due to the user provided inputs) and *bound* at line 9 (due to the $arr[i]-arr[j]$ computation). We denote these two sources leading to the analysis imprecision as alarm cause points[2], $CP_7$ and $CP_9$, respectively. Henceforth in the paper, we use $A_n$, $O_n$, and $Z_n$, respectively, to denote AIOB, OFUF, and ZD alarm at line $n$.

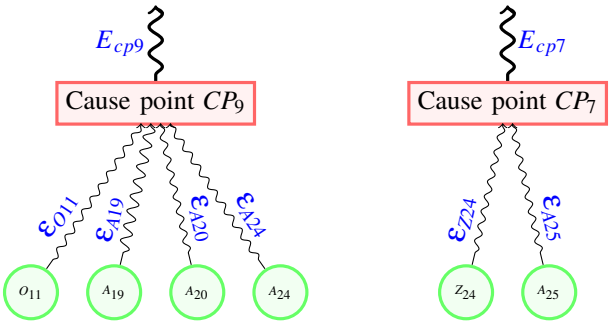


Fig. 2. Abstract representation of effort to inspect the alarms in Figure 1

*1) The Original Approach to Handling Alarms:* The effort spent during a manual inspection of alarms individually varies depending on the code complexity, tools support used, reviewing skills of the user, type of alarms, reporting of alarms, and so on [3]. Thus, we represent manual effort needed to inspect the example alarms using abstractions as shown in Figure 2, where the effort is shown in terms of two components[3]. For an alarm point $\alpha$, the first component denoted as $\varepsilon_\alpha$, represents the effort required to traverse the code from $\alpha$ to its cause point $cp$. The second component, denoted $E_{cp}$, represents the effort in manual inspection to identify the values of the relevant variable(s) at the cause point $cp$.

Note that the effort shown in Figure 2 may also vary depending on the reuse of knowledge gained during the inspection process. For example, assuming $A_{19}$ is inspected earlier, inspecting $A_{20}$ (i.e. effort $\varepsilon_{A20}$) may correspond to checking if the values of $p$ are same at $A_{20}$ and $A_{19}$, eliminating the need of traversal from $A_{20}$ to $CP_9$. Such a reuse is not depicted in Figure 2 for simplicity of modeling. However, we assume there is complete reuse of knowledge in the second component. That is, the identified values of respective variables at the cause points are remembered (say, from the initial inspections of $O_{11}$ and $Z_{24}$) and the values are reused in subsequent inspections of the other alarms. With these assumptions, the total effort required to inspect all the example alarms individually is:

$$\varepsilon_{orig} = ((\varepsilon_{O11} + E_{cp9}) + \varepsilon_{A19} + \varepsilon_{A20} + \varepsilon_{A24}) + ((\varepsilon_{Z24} + E_{cp7}) + \varepsilon_{A25})$$

[1]The alarms generated on the example may vary depending on the analysis domain and reporting methodologies [22].

[2]Throughout this paper, alarm cause points mean the program points that cause imprecision in the analysis and not a root cause for an error.

[3]These components would still be required if alarms are inspected using the sophisticated user interfaces available in the existing analysis tools.

*2) The Proposed Approach of Handling Alarms:* Our approach reports the two causes ($CP_7$ and $CP_9$) and seeks the values of variables *str* and *bound* at these points from the user in order to resolve the alarms. The manual effort spent by the user in identifying these values, say $\mathfrak{E}_{new}$, is $E_{cp7} + E_{cp9}$. A re-analysis of the code using the user-provided input values reports the alarm points either as safe or unsafe depending on the values, leading to generation of no alarm[4]. Thus, the effort saved by our proposed approach is

$$\begin{aligned}\mathfrak{E}_{saved} &= \mathfrak{E}_{orig} - \mathfrak{E}_{new} \\ &= (\mathfrak{E}_{O11} + \mathfrak{E}_{A19} + \mathfrak{E}_{A20} + \mathfrak{E}_{A24}) + (\mathfrak{E}_{Z24} + \mathfrak{E}_{A25})\end{aligned} \tag{1}$$

Observe that the effort saved ($\mathfrak{E}_{saved}$) is not because of an improvement in the precision of the client static analysis, but it is due to the elimination of multiple alarms in one go and elimination of redundant code traversals performed manually from the alarm points to their corresponding cause points. Given the sizes and complexity of the industrial applications, we expect this saving to be considerable.

In Equation 1, $\mathfrak{E}_{saved}$ can be maximized by minimizing $\mathfrak{E}_{new}$. We achieve this by presenting a cause point-specific query seeking abstract information necessary for the safety of alarms (similar to [2], [16]). For example:

1) For $CP_9$, instead of asking *"What are the values computed by 'arr[i] − arr[j]' at line 9?"*, it is more relevant and intuitive to ask *"Does the computed value(s) at line 9 by 'arr[i] − arr[j]' lie in the range of* $\langle 1, 34\rangle$*?"*. The lower (resp. upper) bound is inferred considering the safety of $A_{24}$ (resp. $A_{19}$).
2) For $CP_7$ we ask *"Is the input string 'str' always non-empty and contains less than 20 characters?"*. Here, the requirement of *non-empty* is for the safety of $Z_{24}$ and the requirement on size ($< 20$) is for the safety of $A_{25}$.

The framing of the above queries is described in detail in Section V-A.

In our approach, the cause points are modeled as close as possible to the basic reason generating the alarms. For example, we could have safely modeled $CP_{14}$ (line 14) as the cause point for $Z_{24}$ and $A_{25}$. However, intuitively, answering a question corresponding to $CP_{14}$ would incur more effort as compared to answering the question associated with $CP_7$[5]. This suggests the need of a systematic modeling of cause points. It is described in the next section.

## III. Cause Points Analysis

This section describes modeling of alarm cause points and an analysis to discover them. We begin by describing various types of unknowns observed during the analysis.

### A. Modeling of Unknowns

Achieving satisfactory precision in static analysis is difficult primarily because the values of certain variables are simply unknown statically. Such values have to be treated as nondeterministic choices by a sound analysis tool during the analysis leading to a plethora of alarms [2], [5]–[7]. In their work [2], Dillig et al. have broadly classified these unknown variables into two types: *input variables* denoting unknown values of program inputs, and *abstraction variables* representing unknowns arising due to approximating program behavior. We refine the classification of unknowns depending on the basic operations (origins) leading to them, and it is informally described below restricting the discussion to C language.

***i-unknowns*:** These unknowns arise due to unknown values of program inputs: the values received as inputs from user, read from network or file or sensors, linked at load-time, etc. In Figure 1, *str* in *scanf*() at line 7 is an *i-unknown*.

***c-unknowns*:** These are computational unknowns resulting from approximation of complex computations in the program. For example, a non-linear arithmetic such as $ratio * a * b + size$ is a *c-unknown*. Further, any computation involving two or more unknowns or an unknown due to involvement of a pointer is also modeled as *c-unknown*.

***loop-unknowns*:** We observe that many unknowns (and thus the alarms) in practice are generated due to loops whose bounds (loop execution count) cannot be determined statically. We model this source for unknowns as *loop-unknowns* (described in Section III-B1). For example, the loop at line 10 in Figure 3 is a *loop-unknown*.

***ds-unknowns*:** These unknowns result due to abstracting the values of unbounded data structures like list, stack, queue or even the arrays [5], [23], [24].

***lib-unknowns*:** This category corresponds to the unknowns resulting from calls to library or external functions whose body is not available for analysis. For example, the call $lib(t)$ at line 6 in Figure 3 is a *lib-unknown* assuming the body of *lib* function is not available during the analysis.

***p-unknowns*:** These correspond to not knowing (in)feasibility of certain execution path(s) carrying known but unsafe value to the alarm program point. This is described in Section III-B2.

Note that the origins of unknowns, and hence the above categorization of unknowns, can vary depending on the programming language and also the granularity of modeling. We have modeled them into 6 types for the discussion and experiments. We believe this categorization will apply to other programming languages as well (like C++ and Java) with addition of a few more categories.

### B. Modeling of Cause points

We associate an origin $o$ of unknown values with its program point $p$ that causes a variable $u$ to be an unknown and augment the association with its unknown type $t$. We denote the association by $o^t_p$ and refer to it as a cause point of $u$. The origin $o$ is either an expression or a loop statement, and the unknown type $t \in T = \{i, c, loop, ds, lib, p\}$ corresponds to *i-unknown*, *c-unknown*, *loop-unknown*, *ds-unknown*,

[4]In our approach, the error(s) marked by the client analysis are reported separately and the user handles them differently.

[5]It is with the assumption that the analysis tool is precise in handling strings.

```
1  void foo()
2  {
3    int i, a, b, x, y, z, t, tmp, arr[20];
4    ... // some code
5    x = arr[a]; // assume 0 ≤ a < 20
6    y = lib(t);
7    tmp = 1/x; // {x, Z7} → {arr[a]^ds_5}
8
9    i = 0; t = 5;
10   while(i < 10;)
11     a = arr[t]; // t → {while^loop_10}
12     t=t+1;
13     y=y+1;// {y, O13} → {lib(t)^lib_6, while^loop_10}
14     if(random()%2) i=i+1;
15   }
16   arr[y]=0;// {y, A17} → {lib(t)^lib_6, while^loop_10}
17
18   z = arr[b]; // assume 0 ≤ b < 20
19   t = 1/(z+1); // {z, O19, Z19} → {arr[b]^ds_17}
20 }
```

Fig. 3. Examples of unknowns with their cause points

*lib-unknown*, and *p-unknown* respectively. For brevity, we denote the causal relationship as $u \rightarrow \{o^t_p\}$, where the operator $\rightarrow$ reads "*is caused by*". We use a set on the RHS of $\rightarrow$ because an unknown can be caused by more than one cause point.

Further, a cause point of an unknown $u$ at a program point $p$ is also a cause point for another unknown $v$ at a program point $q$ if $v$ receives its values from $u$ through some path from $p$ to $q$. Thus, only the basic operations leading to unknowns directly or transitively are modeled as cause points. The cause points of unknowns in an alarm expression are referred to as cause points of the alarm. We use the same notation $\rightarrow$ to indicate the alarms to their cause points relationship. The comments in Figure 3 show examples of alarms and their cause points, where the effect of cause points (the unknowns and alarms caused) are shown on the LHS of $\rightarrow$. The program points in the alarm cause points are denoted using line numbers.

*1) Handling of loop-Unknowns:* In this section, we discuss the handling of *loop-unknowns* in more details. Consider a loop $while(p)\{s\}$ whose loop execution count is determined by some unknowns $U_L$ being used in $p$. Let $V_L$ be the set of variables from $s$ whose value(s) are dependent on iteration of the loop. There are two possibilities:

*Possibility 1.* The loop execution count can be expressed in terms of the unknowns in $U_L$. In this case, the cause points of the unknowns in $U_L$ are regarded as the cause points for an unknown in $V_L$. For example, in Figure 1, the execution count of the $for$ loop at line 11 is expressible as the value of another unknown *bound*. Here, $U_L = \{bound\}$, and $V_L$=$\{k\}$. Thus, $k \rightarrow \{(arr[i] - arr[j])^c_9\}$.

*Possibility 2.* The loop execution count cannot be expressed in terms of values of the unknowns in $U_L$. In this case, a new cause point of *loop-unknown* type is created for an unknown in $V_L$. For example, the execution count of the *while* loop at line 10 in Figure 3 can not be known or expressed as values of some other unknown although the loop appears bounded, because the loop variable $i$ is incremented nondeterministically at line 14. For this case, the causality at line 10 is shown as $(V_L = \{i, y, t\}) \rightarrow \{\text{while}^{loop}_{10}\}$.

```
1  void foo(int a){
2    int x=0, y;
3    scanf("%d",&y);
4    if(a == 1){
5     x = 10;
6     y = 1;
7    }
8    if(a == 1){
9     t=1/x;//Z9 → {x=0^p_2}
10   }
11
12   if(x == 10)
13    t=1/y;//Z13 → {scanf(y)^i_3}
14 }
```

Fig. 4. Alarms due to path infeasibility

*2) Path Infeasibility-related Cause Points:* Cause point modeling described in Section III-B is not sufficient for alarms that are generated due to approximations of execution paths by the control flow paths. A simplified example of such an alarm ($Z_9$) is shown in Figure 4. The alarm results due to inability of the analysis to determine the infeasibility of the path in which 0 (known but unsafe value assigned at line 2) reaches the alarm point. Such cause points are categorized to have *p-unknown* as its type. A cause point of this type is denoted as $e^p_n$, which corresponds to not knowing infeasibility of reaching some known but unsafe value assigned in expression $e$ at point $n$ to the alarm program point.

### C. Computation of Cause Points

We compute the alarm cause points using data flow analysis [25], [26] formally presented in the Appendix A. We have presented the analysis at an intraprocedural setting which can be easily lifted to interprocedural setting. For an alarm $\alpha$, let $e$ be the alarm expression with $vars(e)$ being the variables used in it. The cause points of the alarm $\alpha$ at program point $p$, say $C_\alpha$, are identified as below, where $In_p$ represents results computed by the analysis at the start of program point $p$.

$$C_\alpha = \bigcup_{v \in vars(e)} In_p(v) \tag{2}$$

When $C_\alpha$ is empty, the alarm has a cause point of *p-unknown* type. Also, an alarm can have a cause point of *p-unknown* type along with the other types of cause point(s). Reaching definitions [25], [26] can be used to locate cause points of *p-unknown* type, i.e. the program points assigning unsafe value(s).

## IV. Ranking of Cause Points

To improve the effectiveness of manual inspections of alarms further, we prioritize the cause points using three metrics as described below.

### A. Unknown Type-based Ranking

The cause points are ranked based on their unknown type in the order of $i > lib > p > loop > c > ds$, where the LHS of $>$ operator has higher priority over the RHS. This ranking is

based on our hypothesis that (a) alarms caused by *i-unknowns* and *lib-unknowns* are more likely to be *true errors*, and (b) alarms caused by *c-unknowns* and *ds-unknowns* are more likely to be *false alarms*. The following intuitions serve as the basis for the hypothesis.

1) The cause points with *i-unknown* type are the sources for free variables in a program that operates in a partially defined environment, and failure to validate them mostly leads to an error due to certain assumptions made on the input values by the programmer [27]. The validated inputs are no longer unknowns due to the restrictions put during their validations.
2) The implementation of library functions generally adheres to the specification known to the programmer. However, many times programmers miss to validate the values returned by the library function calls.
3) The cause points with *p-unknown* denotes presence of unsafe value, and hence the infeasibility of the value reaching the alarm program point must be ensured. Given the large volume and high complexity of the code in practice, the existence of a path carrying the unsafe value to the alarm program point is not unlikely.
4) Modeling every complex computation and a read of an unbounded data structure as an unknown itself reduces its probability to cause an alarm as an error.

### B. Grouping of Cause Points

The cause points having similar unknown type are grouped depending on their lexical similarity or proximity. When a cause point cannot be grouped with any other cause point, it is treated as the only member of its own group.

*1) Lexical Similarity-based Grouping:* In this approach, lexically similar cause points are grouped together so that their corresponding questions are answered in one go. For example, all the cause points arising due to calls to the same library function (say *readSensor*()) are grouped together presenting an opportunity to frame a query at the group level. Also, in Figure 3, the cause points $arr[a]^{ds}_5$ and $arr[b]^{ds}_{18}$ are grouped as they relate to the same array.

The similarity identification may vary based on the type of unknown. For example, the identifiers representing data structures (resp. library functions) in cause points of *ds-unknown* (resp. *lib-unknown*) type are compared respectively. The entire cause point expression is matched for *c-unknown* and *p-unknown* type cause points. Cause points of *i-unknown* and *loop-unknown* types are excluded from such grouping.

*2) Proximity-based Grouping:* In this approach, the cause points belonging to the same procedure or file are grouped so that their corresponding questions are answered together. This reduces switching between multiple source files and procedures while answering a set of questions.

### C. Contribution Score-based Ranking

Let $C_\alpha$ be the set of cause points for an alarm $\alpha$. When $|C_\alpha| = 1$, $c \in C_\alpha$ is said to cause $\alpha$ *fully*. Otherwise $c \in C_\alpha$ is said to cause $\alpha$ *partially*. Let $S_c$ be the set of alarms that are caused due to a cause point $c$. To measure the contribution of $c$ in causing $S_c$, we define two scores:

$$\textit{full} \text{ contribution score, } \textit{fc-score}(c) = \sum_{\alpha \in S_c,\ |C_\alpha|=1} 1$$

$$\textit{partial} \text{ contribution score, } \textit{pc-score}(c) = \sum_{\alpha \in S_c,\ |C_\alpha|>1} \frac{1}{|C_\alpha|}$$

The total contribution score (*tc-score*) of a cause point $c$ in alarms generation is computed as given below, where a configurable factor $k \geq 2$ is used to weigh the *fc-score* higher than the *pc-score*.

$$\textit{tc-score}(c) = \textit{fc-score}(c) * k \ + \textit{pc-score}(c) \tag{3}$$

The *tc-score* is used to rank the cause points within a group formed (Section IV-B) where, intuitively, cause points with higher score are prioritized over the ones having lesser score. Further, composite *tc-score* is computed for a group as the sum of *tc-scores* of its grouped cause points, and this is used to rank the groups belonging to an unknown-type category.

Some examples[6] of cause points prioritized according to the proposed ranking scheme are given below. The examples also show contribution scores (*tc-scores*) computed with $k = 2$ for each of the cause points and their groups if any.

Figure 1: $scanf(str)^{i}_{7}$=4 $>$ $(arr[i] - arr[j])^{c}_{9}$=8

Figure 3:
$lib(t)^{lib}_{6}$=3 $>$ $while^{loop}_{10}$=3 $>$ $(arr[b]^{ds}_{18}$=4 $> arr[a]^{ds}_{5}$=2)=6

Shifting the focus in manual inspections from alarms to their cause points not only reduces the inspection effort (Section II) but also improves effectiveness of the manual inspections when the resources are limited. For example, alarms that are more likely to be true errors can be identified using cause points types and inspected first (our hypothesis or customizations to it). Further, contribution score can be used as the primary ranking criterion when inspection of more number of alarms is demanded in a given time.

## V. Interactive Static Analysis

This section describes the framing of meaningful cause points-specific queries which is followed by a description of a framework for effective and efficient user interactions with an analysis tool.

### A. Framing Cause Points-specific Queries

Our proposed analysis aims to resolve alarms by accepting inputs from the user for each of the cause points generating the alarms. Thus, a set of cause point-specific queries is presented to the user. To make the user interactions more effective, a query ought to be relevant, more informative, and easy to answer in the context of inspection of alarms. To frame such a query specific to an alarm cause point, we use approach similar to *necessary preconditions* [16] and *proof obligation queries* [2].

[6] $>$ operator denotes higher priority of the left operand over the right operand, while $=$ operator shows contribution score of the left operand.

We frame the queries by identifying required values for unknowns at the cause point required for the safety of the corresponding alarms. That is, the expected values represent necessary precondition on the unknown such that the values never result the alarms into to an error. We perform an analysis to frame the queries by (1) inferring values of the unknown depending on the expression in the alarm and the verification property, and (2) propagating the inferred values of the unknown from the alarm point to the cause point through backward substitution of the values for transitively dependent variables. For example, for $scanf(str)^i_7$ in Figure 1, an important requirement is $length(str)! = 0$ considering the safety of alarm $Z_{26}$. We omit the formalization of this analysis for want of space.

When multiple alarms are caused due to a cause point, the requirement (necessary precondition) on the cause point is computed as the combined requirement necessary for safety of all corresponding alarms. For example, the combined requirement on $CP_9$ in Figure 1, i.e. the *result* computed by $arr[i] - arr[j]^c_9$ is $1 \leq result \leq 34$. It is identified by combining the three requirements $1 \leq result \leq 34$ computed from $A_{26}$, $0 \leq result \leq 34$ computed from $A_{19}$ and $A_{20}$, and $0 \leq result \leq MAX\_INT - 1$ computed from $O_{11}$. Framing queries considering each of the caused alarms separately, as it is done in [2], would generate and present 3 queries for the same cause point $CP_9$, thus increasing the number of queries and user interaction time. Our approach reduces the number of queries presented and this separates our approach from the framing of queries in [2].

The effect of answers to the queries is described below.

**The answer to a query is 'YES':** The next iteration of the client static analysis discharges the corresponding alarms if the *necessary condition* in the query is also *sufficient condition* for the safety of the alarms. For example, answering 'YES' to both the queries in Section II (about the cause points in Figure 1) discharges their corresponding alarms in the next iteration of the client analysis. In the other case, where condition in the query is not *sufficient* to discharge the alarm(s), the input values from this query strengthens the requirements on other cause points that also lead to these alarms.

**The answer to a query is 'NO':** In the next iteration of the client analysis, either the corresponding alarm(s) are marked as true errors or the cause point in the query gets translated into the cause point of *p-unknown* type. For example, answering 'NO' to both the queries associated with the cause points in Figure 1 results its correspondingly caused alarms into errors in the the subsequent client analysis. Further, in Figure 4, answering 'NO' to the query asking '*is user input y non-zero*' associated with the cause point $scanf(y)^i_3$ (inferred from the alarm $Z_{13}$) assigns 0 to $y$ at line 3 in the next analysis iteration. Due to this assignment, the cause point $scanf(y)^i_3$ gets converted to $y{=}0^p_3$ cause point whose corresponding query asks the user the infeasibility of a path carrying this 0 value to the alarm point at line 13. This also indicates resolving an alarm with the proposed approach may involve multiple interleavings of the client analysis and providing user-inputs.

Since the queries are formed based on the expected values, they encourage the user to think of expected behavior of the program thereby improving the understanding of the program state in terms of data invariants at the cause points. This style of questioning allows our approach to relate directly to the analysis generating the alarms and hence makes it more effective.

### B. Interactive Analysis Framework

The alarm cause points are reported using an integrated review framework which is an extended development framework (like Eclipse, NetBeans, and Microsoft Visual Studio) to support code navigation and to interpret the results of the static analysis tool used. The environment extension is to integrate the inspection of alarms seamlessly into their development process, which currently many of the static analysis tools lack today [12]. The path(s) associated with a cause point of *p-unknown* type are shown using path projection [15] and its infeasibility is asked to the user.

*Traceability.* The framework provides traceability (i) from a cause point to the alarms caused by it, and (ii) from an alarm to its cause points. This helps a user to locate alarms generated by a cause point in a query being answered when the answer to the query is 'NO'.

*Customization of the ranking-scheme.* The ranking scheme is allowed to be customized to suit to the requirements of the user, because customizability is an important aspect of a static analysis tool [12], [14]. For example, in certain cases, a user may use the contribution score as the primary ranking criterion or even may change the priorities given to the unknown types.

*Incremental Analysis.* The subsequent iteration of the client analysis (using the user inputs) is performed as an incremental analysis [28] to reduce the wait time between the two consecutive user interactions.

## VI. Evaluation

To determine the practicality and effectiveness of our technique in handling the alarms, we performed an empirical evaluation on a set of real-world applications and our own benchmark. Our evaluation targeted the following research questions:

*RQ*1: What is the reduction in the manual effort using the proposed approach?

*RQ*2: What is the contribution of cause points in generating the alarms and how are they distributed in practice?

*RQ*3: How effective are the metrics used in the ranking of cause points?

### A. Implementation

We selected TCS ECA [21] static analysis tool that analyzes C code for a wide range of verification properties, and used its analysis framework to implement our technique: cause points analysis, framing of queries, and a prototype of the framework described in Section V-B. TCS ECA performs flow-sensitive and context-insensitive interval analysis to compute ranges of the values of program variables. It uses array smashing [10] to scale analysis on real-world applications of large sizes.

### B. Answering RQ1 (Effort Reduction)

*RQ*1 evaluates the suitability and effectiveness of our approach in reducing the alarms inspection cost. To answer the *RQ*1, we performed experiments in industry setting. The experimental setup and results are described below.

*1) Selection of Reviewers:* We selected a mix of 7 reviewers who were users of a static analysis tool with at least one year of experience. Table I presents their total experience with usage of static analysis and their expertise in manual inspection of alarms.

TABLE I
REVIEWER DETAILS

| | Reviewers | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 |
| Experience (Yrs) | 9 | 9 | 9 | 7 | 5 | 3 | 1 | 1 |
| Expertise | H | H | H | M | M | M | L | L |

H = High M = Medium L = Low

*2) Subject Applications:* Table II describes the real-world applications selected in our experiments. They vary in their size, domain, coding language, system completeness, etc.

TABLE II
APPLICATION DETAILS

| Application | Size (KLOC) | Language | System Details |
|---|---|---|---|
| A1 | 45 | C | An automobile embedded system |
| A2 | 30 | C | A mix of embedded systems |
| A3 | 190 | C | A module of an infotainment system |
| A4 | 200 | C | A module of open source Concurrent Versions System (CVS) |
| A5 | 5 | C++ | A Module of an embedded system |
| A6 | 595 | Java | A program analysis workbench |

*3) Static Analysis Tools and Alarms:* Following sound static analysis tools were selected.

1) *TCS ECA*: a static analysis tool for C code.
2) *Polyspace Code Prover* [29], version 9.4 (R2015b): a commercial static analysis tool widely used for analysis of C/C++ code.
3) *NPEDetector* [30]: an open source static analysis tool for detecting null pointer dereferences (NDP) in Java programs.

We selected alarms generated by the above tools corresponding to four most commonly checked verification properties: array index out of bound (AIOB), overflow-underflow (OFUF), division by zero (ZD), and illegal/null dereference of a pointer (IDP/NDP). These were selected because verifying the alarms for these properties requires computation of information governed by data and control flow in the program. Table III shows the alarms selected in our experiments (Column Verification property). Since the manual inspection of alarms is effort-intensive, we avoided selecting alarms from all the four properties together. However, the combinations of the properties, applications, and the tools ensured a variety of alarms.

*4) Cause Points Analysis:* We used TCS ECA (section VI-A) for cause points analysis and framing of the queries in settings 1 to 6. For settings 7 and 8, we manually performed the cause points analysis and framing of queries, because we did not have the tool support for Java/C++ programs and we lacked knowledge in customizing the existing open source static analysis tools available for these languages.

*5) Inspection of Alarms:* The manual inspection of the selected alarms were carried out in the following settings to compute the effort reduction due to our proposed approach. (1) inspection using the *original approach*, where the reviewers were allowed to use any method and tool as per their choice and comfort while inspecting the alarms, and (2) inspection using the cause points-centric approach (*proposed approach*).

Ideally, these two inspections should be performed by different reviewers having the similar expertise and knowledge about the applications (code familiarity). However, identifying such reviewers is difficult. If the same reviewer is used in both the settings, knowledge gained during first inspection affects the effort in the second inspection. Besides, availability of reviewers from industry is limited and costly. Hence we conducted a variety of manual inspections of the alarms as shown in Table III. We used different reviewers (but with similar expertise) in some of the settings to eliminate the effect of knowledge gained during the first inspection. The other settings had the same reviewer in both the inspections. In this case, inspection using the proposed approach is performed first and the order of the two inspections is switched after handling each half of the cause points (and the alarms) to mitigate the effect of knowledge gained in the earlier round.

*6) Results Discussion:* Table III shows effort (time) spent by the reviewers while inspecting the alarms in both the settings, viz. original approach and proposed approach, and the percentage effort reduction. These are respectively shown as $\mathfrak{E}_{orig}$, $\mathfrak{E}_{new}$, and $\mathfrak{E}_{saved}$ (discussed in Section II). The results demonstrate that the proposed approach reduces the manual effort by 23% to 72% depending on the applications, reviewers, analysis tools, and the verification properties. For example, the variation in effort saving in the settings 1 and 2 is due to the change in the reviewers. Further, the reduction in settings 2 and 3 also varied due to the change in properties because inspection of OFUF alarm took longer than inspection of the AIOB alarms. The average effort reduction was by 42%.

The selected applications were well-tested and analyzed using static analysis tools before, hence no error was uncovered during the inspections of selected alarms. Note that this study was to evaluate the effort reduction in industry (real-world) setting due to the proposed approach.

*7) Other Observations:* (1) Table IV also shows maximum execution time (average over 5 different runs) for the original analysis and for an iteration of the proposed analyses. It indicates that the performance overhead in terms of wait time in the user interactions is not much in the proposed analysis. (2) On an average, maximum three iterations (the client

TABLE III
EXPERIMENTAL RESULTS: REDUCTION IN MANUAL EFFORT

| Appli-cations | Tool used | Setting | Verification property (Alarms) | Reviewer(s) | | Manual effort (hours) | | % Effort reduction ($\mathfrak{E}_{saved}$) |
|---|---|---|---|---|---|---|---|---|
| | | | | Original approach | Proposed approach | Original approach ($\mathfrak{E}_{orig}$) | Proposed approach ($\mathfrak{E}_{new}$) | |
| A1(C) | TCS ECA | 1 | AIOB (215) | R1# | R3# | 2.41 | 1.30 | 46.05 |
| | | 2 | AIOB (215) | R6 | R6 | 6.83 | 3.48 | 49.04 |
| | | 3 | AIOB+OFUF+ZD(1000) | R2# | R2# | 9.15 | 6.36 | 30.49 |
| A2 (C) | TCS ECA | 4 | AIOB(196) | R5+R6 | R5+R6 | 3.29 | 2.53 | 23.10 |
| A3 (C) | TCS ECA | 5 | AIOB+ZD(243)* | R7 | R8 | 12.15 | 7.50 | 38.27 |
| A4 (C) | TCS ECA | 6 | IDP (2000)* | R1 | R2 | 2.74 | 1.24 | 54.74 |
| A5 (C++) | Polyspace Code Prover | 7 | AIOB+ZD (85) | R4 | R4 | 3.53 | 2.40 | 32.01 |
| A6 (Java) | NPEDetector | 8 | NDP (555)* | R3# | R2# | 5.68 | 1.58 | 72.18 |

The alarms marked with * are subset of the alarms generated. Reviewers with # are familiar with the code and its functionality.

TABLE IV
DETAILS OF CAUSE POINTS IN APPLICATIONS A1 AND A2

| App. | Verification property | No. of alarms | Alarms caused | | | Type-wise cause point details | | | | | | Total cause points | Analysis time (sec) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Fully | Partially | Interfunc | *i* | *lib* | *p* | *loop* | *c* | *ds* | | Original analysis | Proposed analysis |
| A1 | ZD | 39 | 20 | 19 | 6 | 0 | 0 | 9 | 0 | 6 | 34 | 49 | 62 | 18 |
| | AIOB | 215 | 178 | 37 | 114 | 3 | 1 | 21 | 7 | 57 | 8 | 97 | 102 | 36 |
| | OFUF | 754 | 262 | 492 | 515 | 16 | 0 | 77 | 14 | 390 | 247 | 744 | 197 | 54 |
| | AIOB+ZD+OFUF | 1008 | 460 | 548 | 650 | 16 | 1 | 97 | 16 | 447 | 285 | 862 | 226 | 65 |
| A2 | ZD | 41 | 14 | 27 | 11 | 0 | 0 | 6 | 0 | 9 | 32 | 47 | 63 | 31 |
| | AIOB | 196 | 79 | 117 | 101 | 0 | 5 | 16 | 5 | 100 | 5 | 131 | 72 | 34 |
| | OFUF | 847 | 439 | 408 | 515 | 0 | 13 | 109 | 24 | 683 | 180 | 1009 | 78 | 48 |
| | AIOB+ZD+OFUF | 1084 | 532 | 552 | 627 | 0 | 14 | 121 | 24 | 709 | 211 | 1079 | 85 | 61 |
| Percentage | | | 47.41 | 52.58 | 60.68 | 0.87 | 0.84 | 11.34 | 2.23 | 59.75 | 24.93 | | | |

analysis and user interactions) were required to resolve an alarm. (3) The reviewers in our experiments provided positive feedback about the proposed approach, however generalizing the feedback would require a wider range of experiments using a large number of reviewers and a large set of benchmarks and verification properties.

### C. Answering RQ2

*RQ*2 studies the distribution and contribution of alarm cause points in practice for most commonly checked properties. It is answered by summarizing details about the alarms and their cause points in applications A1 and A2 and comparing them for different properties.

*1) Distribution of Cause Points:* The results in Table IV indicate that the cause points with *c-unknown*, *ds-unknown* and *p-unknown* types are prevalent. About half of the alarms (52%) are caused by more than one cause point (column *Partially*). About 60% of the alarms had at least one cause point located in a function other than the function of the alarm (column *Interfunc*) explaining the effort reduction discussed in Section VI-B6. Further, for 14% of the alarms, the cause points were same as the alarm points due to complex arithmetic operations or arrays used in the alarm expressions (data not shown in the table). We have omitted the details about the grouping of cause points for want of space. Figure 5 compares cause points of alarms in application A1 property-wise. It shows that OFUF alarms have multiple cause points (up to 43) as compared to the alarms related to ZD and AIOB (due to involvement of multiple variables in the OFUF alarms).

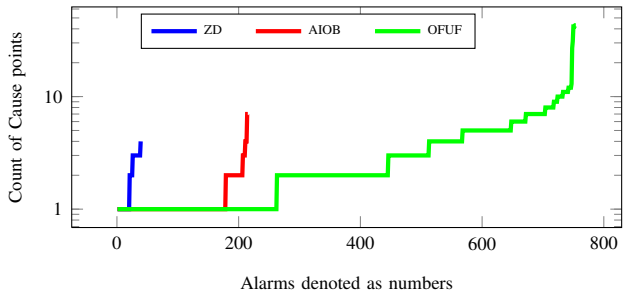


Fig. 5. Property-wise comparison of cause points of alarms on application A1

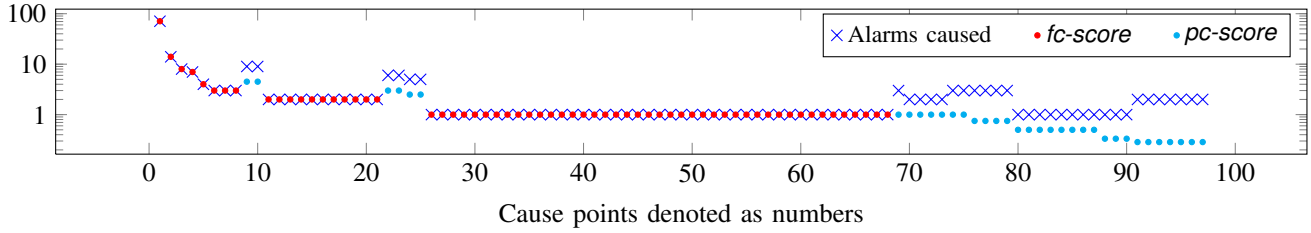


Fig. 6. Contribution of cause points in generation of AIOB alarms on application A1

TABLE V
CAUSE POINTS (UNKNOWNS) TYPE-WISE ERROR-CAUSE RATES

| | Cause point (unknown) types | | | | | |
|---|---|---|---|---|---|---|
| | *i* | *lib* | *p* | *loop* | *c* | *ds* |
| No. of cause points | 7 | 7 | 6 | 16 | 10 | 11 |
| Alarms caused | 11 | 7 | 6 | 56 | 12 | 21 |
| Errors caused | 11 | 7 | 3 | 20 | 0 | 2 |
| % error-cause rate | 100 | 100 | 50 | 35 | 0 | 9 |

*2) Contribution of Cause Points:* Figure 6 shows the cause points of AIOB alarms in application A1, where the cause points are arranged along $X$ axis in the decreasing order of their *tc-scores* computed with $k = 2$. The figure also shows their *fc-score* and *pc-score* along with the alarms caused by them. The figure indicates that about 75% of the alarms are caused by the first 20% of the cause points (close to the 80-20 rule or the Pareto principle). Similar trend has been observed on the other applications as well.

### D. Answering RQ3 (Ranking Effectiveness)

*RQ*3 measures effectiveness of the presented cause points ranking scheme in (a) identifying alarms that are more likely to be true errors, and (b) inspecting a larger number of alarms in a given inspection time. Answer to the latter part is obvious by looking at the contribution of cause points shown in Figure 6 (the 80-20 rule), where the contribution score can be used as the primary ranking criterion. Thus, no further specific experiments were performed for this. The first part is answered by validating our hypothesis used in ranking the unknown types (Section IV). To validate our hypothesis, we computed *error-cause rate* for each of the unknown types[7]. Let $W_u$ be the set of alarms generated due to cause points having $u$ as their unknown type. The *error-cause rate* for an unknown type $u$ is computed as

$$\frac{\text{number of true errors uncovered from alarms in } W_u}{\text{total number of alarms in } W_u} * 100$$

We selected a benchmark consisting of 20 C programs written by 8 software developers during an exercise given for improving their programming skills. The developers had upto two years of experience. On an average the programs had 520 lines of code (max: 1320, min: 87). The programs were well-tested but not verified using any static analysis tool. We used Polyspace code prover[8] to verify the programs, and randomly selected 113 alarms reported for AIOB, ZD, OFUF, and IDP properties. We manually performed cause points analysis and inspected the alarms to compute *error-cause rate* for each of the unknown types. The programs and analysis results are available at https://sites.google.com/site/causepointsanalysis/. Results of this activity are summarized in Table V. The errors for *i-unknown* type were reported because of usage of user provided values (inputs) to index an array (causing AIOB errors). The 7 errors corresponding to *lib-unknown* type were reported because of a failure to check the return values of *malloc* function for *null* value. The results in Table V validate our hypothesis used to rank the alarm cause points, demonstrating effectiveness of the ranking in identifying alarms that are more likely to be errors.

### E. Handling Threats to the Results Validity

*Threat 1 (Selection of reviewers).* A major threat in generalizing our results is the selection of reviewers and methodology used to measure the effort reduction. We tried to mitigate it by selecting a variety of reviewers with different experience, roles, and expertise (Table I) and conducting the inspections in several combinations (Section VI-B). For example, the settings in Table III vary based on reviewers' familiarity about the code and its functionality, because the familiarity influences the amount of effort spent in the inspections.

*Threat 2 (Selection of Alarms).* Manual effort needed to inspect alarms on an application also varies depending on the verification properties, application complexity, and their reporting by an analysis tool, thus posing a threat to validity of our results. To address this, we selected (1) four most commonly checked properties in practice, (2) a mixed set of applications of varying size, complexity, domain, and programming languages, and (3) tools of varying characteristics. Also, we ensured a variety in the alarms and programs (developed by multiple programmers) used to measure effectiveness of the ranking scheme (Section VI-D).

## VII. RELATED WORK

Here, we compare our technique with different classes of the state-of-the-art techniques attempting to reduce the effort in manual inspection of alarms.

[7]We avoided performing experiments with a (fixed) given inspection time because the time required to inspect alarms varies significantly depending on several parameters as shown in Table III.

[8]We preferred Polyspace over TCS ECA due to its higher precision.

*1) Classification of Unknowns:* Several studies like [2], [4], [5] have broadly classified alarm causing unknowns into two categories: *input* variables, and *abstraction* variables. We have extended this classification to several unknown types depending on their origins. It is our first attempt to systematically model the origins of unknowns as the alarms cause points and shift the focus from alarms to them.

*2) Ranking of Alarms:* The existing alarms ranking techniques (surveyed in [31], [32]) classify alarms as actionable and non-actionable alarms, or prioritize them based on the information about the code commit messages, code change history, alarm fix history, user-feedback, etc. Unlike to these techniques that rank the alarms directly, our scheme prioritizes the alarm cause points, resulting in indirect ranking of the alarms. To the best of our knowledge, no existing technique exploits the types of the unknowns to rank the alarms.

*3) Pruning of Alarms:* Blackshear et al. [4] have proposed a technique to suppress alarms generated from overly demonic environments (modular verification) based on semantic inconsistency detection. Mangal et al. [19] have used user's feedback (liking or disliking of a subset of reports) to allow users to tailor analysis precision and cost. *Angelic verification* [17] has been used to constrain an analyzer to report alarms only when no acceptable environment specification exists to prove the property. These techniques help user by pruning the alarms, while our technique simplifies the manual inspections without pruning the alarms.

*4) Framing of Queries:* Our approach of forming the queries is motivated by the work of Cousot et al. [16] that proposes to use *necessary* preconditions over the *sufficient* preconditions. The necessary preconditions are inferred and hoisted to the method entry as required by the design by contract programming methodology, whereas our approach uses them for receiving the user inputs efficiently. The queries proposed by Dillig et al. in [2] are formed during post-processing of alarms and are ranked based on a cost function. Our approach tightly couples the analysis and framing of queries and ranks the queries as per the cause points. Also, our style of formulating the queries reduces the number of queries presented (Section V-A).

*5) Semi-automatic Error Diagnosis:* This class of techniques target improving analysis precision while dealing with the unknowns or providing diagnosis oriented information. Dillig et al. [5] have used systematic reasoning about the unknown values to improve precision and scalability of the analysis. Our technique does not improve analysis precision but reduces the manual inspection effort by locating the causes to the alarms. Rival [6], [9] helped users in judging a given alarm as true error or false alarm by proposing a framework for semi-automatic investigation. The investigation method uses semantic slicing requiring more sophisticated algorithms, whereas our analysis is simpler to implement and aims resolving multiple alarms (that have the same reason) in one go.

*6) Use of Triaging Checklists:* Phang et al. [13] have proposed use of triaging checklists to provide users with a systematic guidance in identifying false alarms. Similar to this, Ayewah et al. [33] have proposed use of a checklist to enable more detailed inspections. The checklists based approach still requires manual inspections to locate causes of an alarm, whereas our technique directly presents the alarm cause point(s) along with informative queries about them.

*7) Use of Novel User-interfaces/Visualization Tools:* Phang et al. have presented a novel user interface toolkit called Path Projection [15] to help users to visualize, navigate, and understand program paths. Further, the commercial static analysis tools like Astreé [34], and CodeSonar [11] provide a visualization tool to support user during the manual inspections. Inspecting the alarms individually using these tools still incurs redundancy for alarms that are caused due to the same cause points. Our approach handles the redundancy problem by resolving such alarms in one go. Further, these visualization tools augment our technique to present the alarm cause points and the paths more effectively.

*8) Similarity-based Grouping of Alarms:* It is widely used technique to identify and group the similar alarms such that only the alarm(s) marked as representative alarm(s) from each group get inspected allowing to skip inspection of other alarms from the group [3], [8], [18], [35]. This reduces the number of alarms to be inspected. However, manual inspection of the representative alarms is still a time-consuming activity as they are inspected individually. Our technique can help to reduce manual effort further by showing the cause points of the representative alarms and presenting queries about them.

The following combinations of different aspects of our technique makes our work novel: modeling of unknowns and alarm cause points, ranking of the cause points, formulating queries about them, and interleaved rounds of the static analysis and seeking user inputs at the alarm cause points. Further, since our technique is orthogonal to many of the existing techniques, we believe that they can be combined together to pool in the complementary strengths.

## VIII. Conclusion

In this paper, we have proposed a static analysis approach centered around the alarm cause points to enable efficient and effective handling of alarms. Our technique includes modeling the sources of unknowns as alarm cause points, their ranking using a pragmatic approach, and framing of cause points-specific queries for efficient user interactions. These aspects allow us to shift the focus in manual inspections from alarms to their cause points, providing the following benefits: (a) reducing the inspection cost, and (b) identifying more errors in a given time. The effectiveness of the proposed technique is demonstrated by our experiments in industry setting, with the average effort reduction of 42%.

The proposed approach is generic and can also yield similar benefits for manual inspections of alarms belonging to other analysis properties, tools and programming languages. Also, our approach can be applied in conjunction with many of the existing alarms handling techniques to complement each other and reduce the manual inspection effort further.

## REFERENCES


[1] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 598–608.

[2] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 181–192.

[3] T. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, Sept 2013, pp. 106–115.

[4] S. Blackshear and S. K. Lahiri, "Almost-correct specifications: A modular semantic framework for assigning confidence to warnings," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 209–218.

[5] I. Dillig, T. Dillig, and A. Aiken, "Reasoning about the unknown in static analysis," *Commun. ACM*, vol. 53, no. 8, pp. 115–123, Aug. 2010.

[6] X. Rival, "Understanding the origin of alarms in astrée," in *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, 2005, pp. 303–319.

[7] T. Muske, "Improving review of clustered-code analysis warnings," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 October 3, 2014*. IEEE, 2014, pp. 569–572.

[8] Z. Fry and W. Weimer, "Clustering static analysis defect reports to reduce maintenance costs," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 282–291.

[9] X. Rival, "Abstract dependences for alarm diagnosis," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, ser. APLAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 347–363.

[10] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 196–207.

[11] R. P. Jetley, P. L. Jones, and P. Anderson, "Static analysis of medical device software using CodeSonar," in *Proceedings of the 2008 Workshop on Static Analysis*, ser. SAW '08. New York, NY, USA: ACM, 2008, pp. 22–29.

[12] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[13] K. Y. Phang, J. S. Foster, M. Hicks, and V. Sazawal, "Triaging checklists: a substitute for a PhD in static analysis," in *PLATEAU 2009*.

[14] L. Layman, L. Williams, and R. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, Sept 2007, pp. 176–185.

[15] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '08. New York, NY, USA: ACM, 2008, pp. 57–63.

[16] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo, "Automatic inference of necessary preconditions," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds. Springer Berlin Heidelberg, 2013, vol. 7737, pp. 128–148.

[17] A. Das, S. Lahiri, A. Lal, and Y. Li, "Angelic verification: Precise verification modulo unknowns," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9206, pp. 324–342.

[18] D. Zhang, D. Jin, Y. Gong, and H. Zhang, "Diagnosis-oriented alarm correlations," in *Proceedings of the 2013 20th Asia-Pacific Software Engineering Conference (APSEC) - Volume 01*, ser. APSEC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 172–179.

[19] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 462–473.

[20] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22–29, Sept 2008.

[21] TCS Embedded Code Analyzer (TCS ECA), http://www.tcs.com/engineering-services/Pages/TCS-Embedded-Code-Analyzer.aspx, [Online accessed 20-Aug-2016].

[22] T. Muske and P. Bokil, "On implementational variations in static analysis tools," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, March 2015, pp. 512–515.

[23] D. Monniaux and F. Alberti, "A simple abstraction of arrays and maps by program translation," in *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, S. Blazy and T. Jensen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 217–234.

[24] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The essence of computation," T. A. Mogensen, D. A. Schmidt, and I. H. Sudborough, Eds. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Design and Implementation of a Special-purpose Static Program Analyzer for Safety-critical Real-time Embedded Software, pp. 85–108.

[25] U. Khedker, A. Sanyal, and B. Karkare, *Data Flow Analysis: Theory and Practice*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 2009.

[26] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[27] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 87–97.

[28] R. Mudduluru and M. Ramanathan, "Efficient incremental static analysis using path abstraction," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, S. Gnesi and A. Rensink, Eds. Springer Berlin Heidelberg, 2014, vol. 8411, pp. 125–139.

[29] "Polyspace code prover," http://in.mathworks.com/products/polyspace-code-prover/, [Online: accessed 20-Aug-2016].

[30] NPEDetector, http://sourceforge.net/projects/npedetector/, [Online: 20-Aug-2016].

[31] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363 – 387, 2011.

[32] ——, "A model building process for identifying actionable static analysis alerts," in *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*, April 2009, pp. 161–170.

[33] N. Ayewah and W. Pugh, "Using checklists to review static analysis warnings," in *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, ser. DEFECTS '09. New York, NY, USA: ACM, 2009, pp. 11–15.

[34] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The Astreé analyzer," in *Programming Languages and Systems*, ser. Lecture Notes in Computer Science, M. Sagiv, Ed. Springer Berlin Heidelberg, 2005, vol. 3444, pp. 21–30.

[35] W. Lee, W. Lee, and K. Yi, "Sound non-statistical clustering of static analysis alarms," in *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 299–314.

## APPENDIX

### A. Data flow analysis for cause points computation

Let $\mathsf{N}$ (resp. $\mathsf{E}$) be the set of nodes (resp. expressions and loop statements) in the control flow graph of the program being analyzed, and $\mathsf{V}$ be the set of variables in the program. Let $\mathsf{T} = \{i, c, loop, ds, lib, p\}$ be the set of unknown types. We denote a cause point using a tuple $\langle e, n, t \rangle$, where $e \in \mathsf{E}, n \in \mathsf{N}, t \in \mathsf{T}$.

We use the following notational conventions:

- $\mathsf{C} = \mathsf{E} \times \mathsf{N} \times \mathsf{T}$ is the set of cause points in the program.
- Mapping $\mathsf{cpoints} = \mathsf{V} \mapsto 2^{\mathsf{C}}$ relates a variable $v \in \mathsf{V}$ to its potential cause points $cp \subset 2^{\mathsf{C}}$.
- $\alpha$ ranges over the set $A = 2^{\mathsf{cpoints}}$ and represents a mapping from variables to their cause points. $\alpha(v)$ returns the set of cause points of variable $v$ and $\alpha[v \mapsto c]$ updates the cause points of $v$ to the set $c$ in the mapping $\alpha$.
- $pred(n)$ returns predecessors of a given node $n \in \mathsf{N}$.
- *StartNode* denotes the starting node in intraprocedural CFG (control flow graph) of a function.
- $input(v)$ denotes a function accepting values for $v$ from the user, environment, or the files.
- $lib(...)$ denotes calls to library functions.

*1) Lattice:* Our analysis computes subsets of $\mathsf{cpoints}$ flow-sensitively at each node $n \in \mathsf{N}$. We use $\mathit{In}_n$ and $\mathit{Out}_n$ to denote the values computed at the start and exit of the node $n$ respectively. The lattice of these values is $\langle A = 2^{\mathsf{cpoints}}, \sqcap_A \rangle$. As $\mathsf{cpoints} = \mathsf{V} \mapsto 2^{\mathsf{C}}$ is defined in terms of lattice $(2^C, \cup)$, the meet operation $\sqcap_A$ is defined as shown below. Given $x, y \in A$:

$$x \sqcap_A y = \left\{ \left(v,\ \left(c \cup c'\right)\right) \mid (v,c) \in x, \left(v,c'\right) \in y \right\} \tag{4}$$

Let $c1, c2 \in \mathit{Constants}$; $m, n \in \mathsf{N}$; $u, v, w \in \mathsf{N}$; and $e, e1, expr \in \mathsf{E}$

$$\mathit{In}_n = \begin{cases} \{\ \} & n = StartNode \\ \prod\limits_{p \in pred(n)}^{A} \mathit{Out}_p & \text{otherwise} \end{cases} \tag{5}$$

$$\mathit{Out}_n = \begin{cases} \mathit{X}[v \mapsto \{\langle input(v), n, i \rangle\}] & n : input(v) \\ \mathit{Expr}_n(\mathit{X}, v, expr) & n : v = expr \end{cases} \tag{6}$$

$$\mathit{Expr}_n(\mathit{X}, v, e) = \begin{cases} \mathit{X}[v \mapsto \{\ \}] & e : c1 \quad \text{or } e : c1 \oplus c2 \\ \mathit{X}[v \mapsto \mathit{X}(u)] & e : u \\ \mathit{X}[v \mapsto \{\langle e, n, ds \rangle\}] & e : u[v] \\ \mathit{X}[v \mapsto \{\langle e, n, lib \rangle\}] & e : lib(...) \\ \mathit{Expr}_n(\mathit{X}, v, e1) & e : c1 \oplus e1 \quad \text{or} \quad e : e1 \oplus c1 \\ \mathit{X}[v \mapsto \{\langle e, n, c \rangle\}] & e : u \oplus w, \quad \mathit{X}(u) \neq \phi \\ & \text{and } \mathit{X}(w) \neq \phi \\ \mathit{X}[v \mapsto \mathit{X}(u) \cup \mathit{X}(w)] & \text{otherwise} \end{cases} \tag{7}$$

Fig. 7. Data flow equations for computing cause points of unknowns.

*2) Data Flow Equations:* Figure 7 shows the data flow equations to compute cause points of unknowns. For simplicity of the equations, the handling of *loop-unknowns* (Section III-B1) and function calls is not shown. The expression *expr* in Equation 6 is assumed to have at most one operator and does not cause side effects.

Note that the formalization does not cover computation of cause points of *p-unknown* type (Section III-B2) as they need to be computed differently to identify the paths carrying known but unsafe values. This computation needs to refer to alarms while the above formalization is independent of the alarms.