Reducing Static Analysis Alarms based on Non-impacting Control Dependencies

Tukaram Muske[1], Rohith Talluri[1], and Alexander Serebrenik[2]

TRDDC, TCS Research, Pune, India
{t.muske, rohith.talluri}@tcs.com
Eindhoven University of Technology, Eindhoven, The Netherlands
a.serebrenik@tue.nl

Abstract. Static analysis tools help to detect programming errors but generate a large number of alarms. Repositioning of alarms is recently proposed technique to reduce the number of alarms by replacing a group of similar alarms with a small number of newly created representative alarms. However, the technique fails to replace a group of similar alarms with a fewer representative alarms mainly when the immediately enclosing *conditional statements* of the alarms are different and not nested. This limitation is due to conservative assumption that a conditional statement of an alarm may prevent the alarm from being an error.
To address the limitation above, we introduce the notion of *non-impacting control dependencies* (NCDs). An NCD of an alarm is a transitive control dependency of the alarm's program point, that does not affect whether the alarm is an error. We approximate the computation of NCDs based on the alarms that are similar, and then reposition the similar alarms by considering the effect of their NCDs. The NCD-based repositioning allows to merge more similar alarms together and represent them by a small number of representative alarms than the state-of-the-art repositioning technique. Thus, it can be expected to further reduce the number of alarms.
To measure the reduction obtained, we evaluate the NCD-based repositioning using total 105,546 alarms generated on 16 open source C applications, 11 industry C applications, and 5 industry COBOL applications. The evaluation results indicate that, compared to the state-of-the-art repositioning technique, the NCD-based repositioning reduces the number of alarms respectively by up to 23.57%, 29.77%, and 36.09%. The median reductions are 9.02%, 17.18%, and 28.61%, respectively.

1 Introduction

Static analysis tools help to automatically detect common programming errors like *division by zero* and *array index out of bounds* [2, 3, 5, 33] as well as help in certification of safety-critical systems [6, 10, 17]. However, these tools report a large number of alarms that are warning messages notifying the tool-user about potential errors [11, 15, 22, 29, 31]. Partitioning the alarms into true errors and false alarms (false positives) requires manual inspection [11, 19, 30]. The large

number of false alarms generated and effort required to analyze them manually have been identified as primary reasons for underuse of static analysis tools in practice [4, 7, 15, 19].

Clustering is commonly used to reduce the number of alarms reported to the user [14, 26]. State-of-the-art clustering techniques [13, 20, 24, 34] group similar alarms[1] together such that (1) there are few dominant and many dominated alarms; and (2) when the dominant alarms of a cluster are false positives, *all the alarms in the cluster* are also false positives. The techniques count only the dominant alarms as the alarms obtained after the clustering.

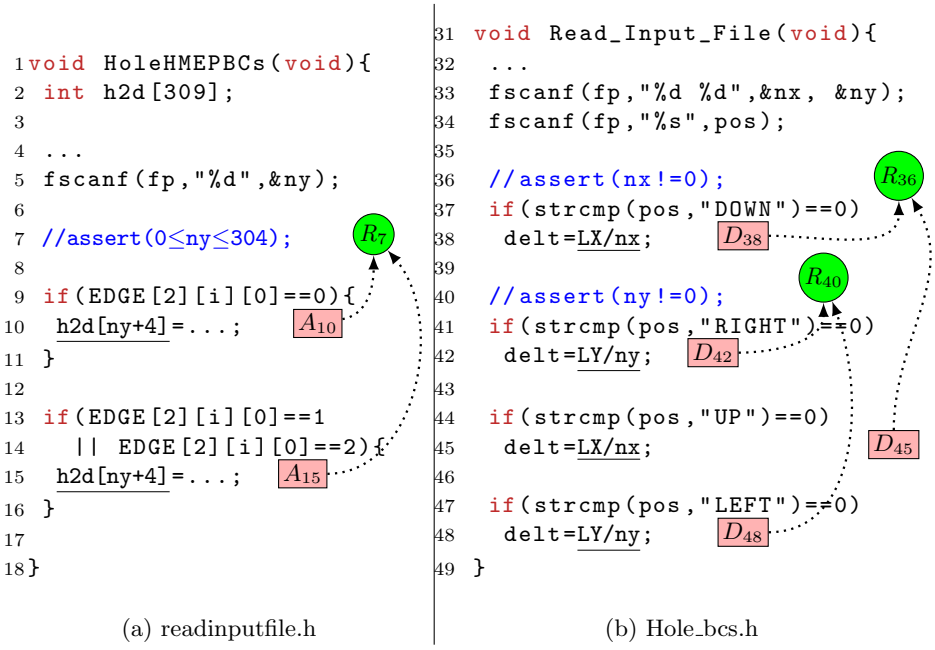
Repositioning of alarms [27] is recently proposed technique to overcome limitations of the clustering techniques [13, 20, 24, 34]. To achieve the reduction in alarms, the technique repositions a group of similar alarms to a program point where they can be *safely replaced* by a fewer newly created representative alarms (called as *repositioned alarms*). The alarms repositioning is safe and performed only if the following *repositioning criterion* is met—when a repositioned alarm is a false positive, its corresponding *original alarms* are also false positives, and vice versa. Thus, the repositioned alarms act as dominant alarms for the original (similar) alarms that are replaced by them.

Problem The alarms repositioning technique [27] described above fails to reposition and merge similar alarms when their immediately enclosing conditional statements are different and not nested. As a consequence, in these cases, the repositioning technique does not reduce the number of alarms. We call these cases *repositioning limitation scenarios*. We illustrate this limitation using the alarms (red rectangles) shown in Figure 1. The code is excerpted from archimedes-0.7.0. The two code examples shown in Figures 1a and 1b are independent of each other. Analyzing the code in Figure 1a (resp. Figure 1b) using any static analysis tool generates two (resp. four) alarms corresponding to *array index out of bounds* (resp. *division by zero*). Grouping these alarms using the state-of-the-art clustering techniques [13, 20, 24, 34] does not reduce their number.

Among the six alarms shown in Figure 1, there exist three groups of similar alarms: A_{10} and A_{15}, D_{38} and D_{45}, and D_{42} and D_{48}. The alarms repositioning technique cannot determine whether the control dependencies[2] (i.e. the enclosing conditional statements) of these alarms *can prevent* the alarms from being an error. Thus, the technique conservatively assumes that the control dependencies of these alarms can prevent the alarms from being an error, i.e., the dependencies *can impact* those alarms. For example, the values read for nx at line 33 can be zero due to which two similar alarms D_{38} and D_{45} get generated. However, the technique conservatively assumes that the control dependencies of these alarms can prevent the zero value read for nx from reaching to lines 38 and 45. As a result of the conservative assumption, the repositioning criterion cannot be guaranteed for repositioning of these two similar alarms (D_{38} and D_{45}) to any

[1] Broadly, two alarms are said to be *similar* if the property/condition checked in one alarm implies the property/condition checked in the other alarm (Section 2).

[2] A control dependency of a program point p is a conditional edge in the control flow graph [1], that decides whether p is to be reached or not (see Section 2).



(a) readinputfile.h (b) Hole_bcs.h

Fig. 1: Examples of alarms to illustrate their NCD-based repositioning.

program point, e.g., to line 36. That is, the resulting repositioned alarm can be an error while none of these two alarms is an error. Thus, the repositioning technique fails to reposition and merge these two similar alarms together. Similarly, the technique also fails to reposition the other two groups of similar alarms shown in Figure 1. As a result, the repositioning technique does not reduce the number of alarms shown in Figure 1.

We find that the above assumption about the control dependencies of the alarms' program points limits the reduction achieved by the repositioning technique, because not every control dependency of an alarm's program point can prevent the alarm being an error. For example, the conditions corresponding to the control dependencies of the alarms shown in Figure 1 are *most likely* to determine whether the program points of those alarms are to be reached and not to prevent the alarms from being an error.

Our pilot study on 16 open source applications indicates that, 38% of the alarms reported after their repositioning are still similar and appear in the repositioning limitation scenarios. These results suggest the scope for improvement.

Our Solution To overcome the problem above and further reduce the number of alarms, we introduce the notion of *non-impacting control dependencies* (NCDs). An NCD of an alarm is a transitive control dependency of the alarm's program point, that does not affect whether the alarm is an error. As we intend to reposition and merge more similar alarms together for reducing their number,

we restrict the scope of NCDs computation to the similar alarms only. Since determining whether a control dependency is an NCD is undecidable, we compute the NCDs of similar alarms approximately (described in Section 4.2). The NCDs computed are subsequently used to reposition the similar alarms by considering the effect of their NCDs (*NCD-based repositioning*). Thus, NCD-based repositioning allows to reposition more similar alarms together and replace them by a fewer repositioned (dominant) alarms than the state-of-the-art repositioning technique. For example, our approach to compute NCDs, identifies the control dependencies of the alarms shown in Figure 1 as NCDs. Repositioning each group of the similar alarms using the NCDs allows to replace the group by a newly created dominant alarm (shown using green circles). Thus, NCD-based repositioning reduces the number of alarms by three.

Although NCD-based repositioning is performed based on approximated NCDs, the repositioned alarms do not miss detection of an error uncovered by the original alarms. Thus, NCD-based repositioning can be expected to further safely reduce the overall number of alarms.

To measure the reduction obtained, we evaluate NCD-based repositioning on total 105,546 alarms generated for the following kinds of applications: (i) 16 open source C applications (ii) 11 industry C applications; and (iii) 5 industry COBOL applications. The alarms are generated by a commercial tool for five safety properties. The evaluation results indicate that, compared to the state-of-the-art repositioning technique, NCD-based repositioning reduces the number of alarms on these applications, respectively by up to 23.57%, 29.77%, and 36.09%. The median reductions are 9.02%, 17.18%, and 28.61%, respectively.

Following are the key contributions of our work.

1. The notion of NCDs of alarms and computing them for similar alarms.
2. An NCD-based repositioning technique to reduce the number of alarms.
3. A large-scale empirical evaluation of the NCD-based repositioning technique using 105,546 alarms on 16 open source and 16 industry applications.

Paper outline. Section 2 presents terms and notations that we use throughout the paper. Section 3 describes the pilot study. Section 4 describes the notion of NCDs and NCD-based repositioning. Section 5 presents a technique/algorithm to implement NCD-based repositioning. Section 6 discusses our empirical evaluation. Section 7 presents related work, and Section 8 concludes.

2 Terms and Notations

Control Flow Graph A control flow graph (CFG) [1] of a program is a directed graph $\langle \mathcal{N}, \mathcal{E} \rangle$, where $\mathcal{N}$ is a set of nodes representing the program statements (e.g., assignments and conditional statements); and $\mathcal{E}$ is a set of edges such that an edge $\langle n, n' \rangle$ represents a possible flow of program control from $n \in \mathcal{N}$ to $n' \in \mathcal{N}$ without any intervening node. We use $n \rightarrow n'$ to denote an edge from node n to node n'. Depending on whether the program control flows conditionally or unconditionally along an edge, the edge is labeled either as conditional

or unconditional. We denote the condition corresponding to a conditional edge $u \rightarrow v$ as $cond(u \rightarrow v)$. A CFG has two distinguished nodes *Start* and *End*, representing the *entry* and *exit* of the corresponding program, respectively. For a given node n, we use $pred(n)$ to denote predecessors of n in the graph.

Except for the *Start* and *End* nodes, we assume that there is a one-to-one correspondence between the CFG nodes and their corresponding program statements. Thus, we use the terms statement and node interchangeably. Henceforth, in code examples we use n_m to denote the node of a program statement at line m. For the sake of simplicity, we assume that the program statements do not cause side effects and the conditional statements (branching nodes) do not update values of a variable.

Program Points We write $entry(n)$ and $exit(n)$ to denote the *entry* and *exit* of a node n, i.e., the program points *just before* and *immediately after* the execution of statement corresponding to the node n, respectively. The entry or exit of a node is assumed not to be shared with entry or exit of any other node even though they may indicate the same program point/state. A program point p_1 *dominates* a program point p_2 if every path from the *program entry* to p_2 contains p_1. A program point p_1 *post-dominates* a program point p_2 if every path from p_2 to the *program exit* contains p_1.

Data Dependencies A variable v at a program point p is said to be *data dependent* on a definition d of v, if d is a reaching definition [16, 28] of v at p. Data dependencies of a variable v are the definitions on which v is data dependent.

Control Dependencies A node w is said to be *control dependent* on a conditional edge $u \rightarrow v$ if w post-dominates v; and if $w \neq u$, w does not post-dominate u [9, 12]. Control dependencies of a node n or a program point $entry(n)$ (or $exit(n)$) are the conditional edges on which the node n is control dependent. A conditional edge e is called as *transitive control dependency* of a point p if e belongs to the transitive closure of control dependencies of p. We use $e \rightsquigarrow p$ to denote that e is a transitive control dependency of a program point p. We say that the conditions of two conditional edges e_1 and e_2 are *equivalent* if $cond(e_1) \Leftrightarrow cond(e_2)$. In the other case, we say that the conditions of the two dependencies are different. On similar lines, we call two conditional edges $n \rightarrow n'$ and $m \rightarrow m'$ *condition-wise equivalent* only if (1) their conditions are equivalent; and (2) every variable in their conditions has same data dependencies at $exit(n)$ and $exit(m)$.

Static Analysis Alarms A static analysis tool reports an alarm at the location where the run-time error corresponding to the alarm is likely to occur. We refer to the tool generated alarms as the *original alarms* and to their locations as the *original locations*. We use $cond(\phi)$ to denote *alarm condition* of an alarm ϕ, i.e., the check performed by the analysis tool for detecting an error. The alarm condition holds *iff* the corresponding alarm is a false positive. For example, $nx \neq 0$ is the alarm condition of the alarms D_{38} and D_{45} shown in Figure 1b. We use *safe values* (resp. *unsafe values*) to refer to the set of values of the variable(s) in $cond(\phi)$ due to which ϕ is a false positive (resp. an error).

We call two alarms ϕ and ϕ' *similar* if $cond(\phi) \Rightarrow cond(\phi')$ or $cond(\phi') \Rightarrow cond(\phi)$. An alarm ϕ is said be a dominant alarm of an alarm ϕ' only if when ϕ is a false positive, ϕ' is also a false positive. We use ϕ_p to denote an alarm ϕ located at a program point p, and thus we say that the transitive control dependencies of ϕ_p are same as the transitive control dependencies of p. We write $e \rightsquigarrow \phi$ to indicate that e is a transitive control dependency of an alarm ϕ. We use tuple $\langle c, p \rangle$ to denote a repositioned alarm at p with c as its alarm condition.

3 Pilot Study

As a sanity check we performed a study to measure (1) what percentage of alarms resulting after the state-of-the-art repositioning [27] are similar; and (2) what percentage of these similar alarms appear in the repositioning limitation scenarios (Section 1). The similar alarms appearing in those limitation scenarios are *candidates* for reducing their number through NCD-based repositioning.

We selected 16 open source C applications that were previously used as benchmarks for evaluating the alarms clustering techniques [21,34] and the repositioning technique [27]. We analyzed these applications using our commercial static analysis tool, TCS ECA [32], for five safety properties: division by zero, array index out of bounds (AIOB), arithmetic overflow and underflow, dereference of a null pointer, and uninitialized variables. The generated alarms are postprocessed using the clustering techniques [21, 24] and then the resulting *dominant alarms* are repositioned using the state-of-the-art technique [27].

We first identified groups of similar alarms from 64779 alarms generated by the setup above. Next we identified similar alarms in each group that have same data dependencies for their variables, and counted those alarms as the similar alarms appearing in the repositioning limitation scenarios. Results of this study are shown in Table 1. The column *Total Alarms* shows the number of total alarms generated by state-of-the-art grouping and repositioning techniques for the selected five properties. The column *% Similar Alarms* presents percentage of similar alarms in the total alarms. The column *% Same DDs* (resp. *% Different DDs*) presents percentage of the similar alarms that have *same data dependencies* (resp. *different data dependencies*).

The study indicates that, on an average, 50.89% of the alarms obtained after the state-of-the-art repositioning are similar, and 74% of these similar alarms—38% of the total alarms—appear in the repositioning limitation scenarios. Based on these results we expect postprocessing alarms using NCD-based repositioning can help to reduce their number.

4 NCDs of Similar Alarms

4.1 The Notion of NCD of an Alarm

Definition 1 (NCD of an alarm). *Let ϕ be an alarm reported in a program P, and $(n \rightarrow n')$ is a transitive control dependency of ϕ. Let P' be obtained from*

Table 1: Distribution of similar alarms in alarms reported after state-of-the-art clustering and repositioning techniques.

Application	Total Alarms	% Similar Alarms	% Same DDs	% Different DDs
archimedes-0.7.0	2275	51.60	34.24	65.76
polymorph-0.4.0	25	28.00	100.00	0.00
acpid-1.0.8	25	44.00	45.45	54.55
spell-1.0	71	25.35	44.44	55.56
nlkain-1.3	319	53.92	36.63	63.37
stripcc-0.2.0	229	66.81	84.31	15.69
ncompress-4.2.4	92	51.09	53.19	46.81
barcode-0.96	1064	47.09	61.68	38.32
barcode-0.98	1310	46.64	60.72	39.28
combine-0.3.3	819	66.42	71.14	28.86
gnuchess-5.05	1783	51.65	55.27	44.73
antiword-0.37	613	32.79	60.70	39.30
sudo-1.8.6	7433	43.72	54.18	45.82
uucp-1.07	2068	51.50	70.23	29.77
ffmpeg-0.4.8	45137	51.99	82.33	17.67
sphinxbase-0.3	1516	54.68	49.70	50.30
Total	**64779**	**50.89**	**74.55**	**25.45**

P by replacing the condition of the branching node n with a non-deterministic choice function. We say that the dependency $n \to n'$ *is an* impacting control dependency *(ICD) of* ϕ *only if* ϕ *is a false positive in P but an error in P'. Otherwise, say that the dependency* $n \to n'$ *is a* non-impacting control dependency *(NCD) of* ϕ. □

We illustrate the notion of NCD/ICD by categorizing the effect of a control dependency $e \rightsquigarrow \phi_p$ on ϕ_p, where $e = n \to n'$. The classification is based on the values that can be assigned to variables in $cond(\phi_p)$.

Class 1 The variables in $cond(\phi_p)$ are assigned with *safe values* by their data dependencies, and thus ϕ_p is a false positive. In this case, e is an NCD of ϕ_p: replacing the condition of the branching node n—the source node of e—by a non-deterministic choice function does not cause ϕ_p to be an error.

Class 2 The variables in $cond(\phi_p)$ are assigned with *unsafe values* by their data dependencies, and ϕ_p is an error if the unsafe values reach the alarm program point p. In this case, the effect of e on ϕ_p is in one of the following two ways depending on whether the unsafe values reach ϕ_p.

Class 2.1: The condition $cond(e)$ prevents the flow of the unsafe values from reaching ϕ_p and thus ϕ_p is a false positive. In this case, if the condition of the source node n of e is replaced by a non-deterministic choice function, the alarm is an error as those unsafe values reach ϕ_p. That is, e affects whether ϕ_p is an error or a false positive. Thus, in this case, we say that e is an

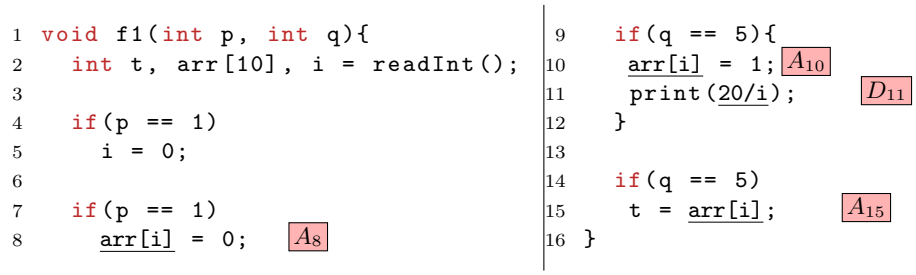


Fig. 2: Examples to illustrate ICDs and NCDs of alarms.

ICD of ϕ_p, and $cond(e)$ is a *safety condition* for ϕ_p because e prevents the alarm from being an error. For example, in Figure 2, the control dependency $n_7 \rightarrow n_8$ of A_8 is ICD.

Class 2.2: The condition $cond(e)$ does not prevent the flow of the unsafe values from reaching ϕ_p and thus ϕ_p is an error. In this case, if the condition of the source node n of e is replaced by a non-deterministic choice function, the alarm would still remain as an error. That is, e does not affect whether ϕ_p is an error or a false positive. Thus, we say that e is an NCD of ϕ_p. For example, in Figure 2, the control dependency $n_9 \rightarrow n_{10}$ of D_{11} is NCD.

4.2 Computation of NCDs of Similar Alarms

Computing whether a given dependency e of an alarm ϕ is an ICD or NCD includes determining whether ϕ is a false positive. As determining whether ϕ is a false positive is undecidable in general [11, 22], determining whether e is an ICD/NCD of ϕ is also undecidable. Thus, we compute approximation of ICDs/NCDs. As we aim to reposition similar alarms together, we focus on computing NCDs of those similar alarms only. For a given set of similar alarms Φ_S and $\phi \in \Phi_S$, the approximation of NCDs/ICDs of ϕ is described below.

Computation of ICDs For an alarm ϕ, we compute its transitive control dependency $e \rightsquigarrow \phi$ as ICD, only if every path reaching each alarm $\phi'_p \in \Phi_S$ has a dependency $e' \rightsquigarrow \phi'_p$ on it such that e and e' are *condition-wise equivalent*. For example, the control dependencies of the similar alarms A_{10} and A_{15} in Figure 2 are ICDs.

Computation of NCDs For an alarm ϕ, we compute its transitive control dependency $e \rightsquigarrow \phi$ as NCD, if there exists a path reaching at an alarm $\phi'_p \in \Phi_S$ without having a dependency $e' \rightsquigarrow \phi'_p$ on it such that e and e' are *condition-wise equivalent*. For example, in Figure 1, the control dependencies of the similar alarms D_{38} and D_{45} are NCDs.

In other words, when $\phi \in \Phi_S$, $e \rightsquigarrow \phi$, and a condition equivalent to $cond(e)$ appears on every path to each of the similar alarms Φ_S, then we treat $cond(e)$ as a *potential safety condition* for each alarm in Φ_S, and thus e as an ICD of ϕ. Otherwise, e is an NCD of ϕ.

Intuition Behind the Approximation The NCDs of similar alarms computed above approximate NCDs as defined in Definition 1. The idea of the approximation is based on the earlier observation by Kumar et al. [18] that removing statements which merely control reachability of an alarm's program point *rarely affects* whether the alarm is false positive or not: removing the non-value impacting control statements of the alarms changed only 2% of the false positive alarms into errors. This suggests that for a given dependency $e \rightsquigarrow \phi$, $cond(e)$ is *rarely* a safety condition for ϕ, i.e., e is *rarely* an ICD of ϕ. Thus, *intuitively, the chance of existing different safety condition for each of the alarms in Φ_S is even lower*: if there exists a safety condition to prevent an alarm from being an error, an *equivalent condition* also should exist for every other similar alarm. For example, in Figure 1, if the condition $strcmp(pos, \text{“}DOWN\text{”}) == 0$ is a safety condition for D_{38}, the same condition should also have been for its similar alarm D_{45}. Thus, we approximate the control dependencies of those two alarms to be NCDs. On similar lines, the control dependencies of the other alarms in Figure 1 are NCDs.

In the next section we discuss that, although the above computation of NCDs is observation-based and approximated, the NCDs computed can be *safely* used to reduce the overall number of alarms.

4.3 NCD-based Repositioning of Similar Alarms

To overcome the limitation of the state-of-the-repositioning (Section 1), we reposition a group of similar alarms by considering the effect of their NCDs. We design NCD-based repositioning to satisfy the following constraints, where R is the set of alarms resulting from the repositioning of a set of similar alarms Φ_S.

$C1$: The program points of the repositioned alarms R together dominate the program point of every alarm $\phi \in \Phi_S$, so that when the repositioned alarms R are false positives, the original alarms Φ_S are also false positives.

$C2$: All the paths between the repositioned alarms R and every alarm $\phi \in \Phi_S$ does not have an ICD of ϕ (that is, all the control dependencies of an alarm $\phi \in \Phi_S$ along a path between the repositioned alarms R and ϕ are NCDs).

$C3$: The number of the repositioned alarms R is strictly not greater than the number of original alarms Φ_S.

The constraint $C1$ ensures that when $\phi \in \Phi_S$ is an error, at least one of the repositioned alarms R is also an error. Thus, the repositioning is *safe*, and the repositioned alarms R together act as *dominant alarms* of the original alarms Φ_S. However, as the repositioned alarms are newly created, with $C1$ we cannot guarantee that when a repositioned alarm $r_p \in R$ is an error, at least one of its corresponding original alarms $\Phi' \subseteq \Phi_S$ is an error. That is, r_p may detect an error spuriously. The spurious error detection occurs only when every path between r_p and each $\phi \in \Phi'$ has an ICD of ϕ.

To overcome the problem above—a repositioned alarm detecting a spurious error—we add the second constraint $C2$. The constraint $C1$ together with

$C2$ guarantees that when a repositioned alarm is an error, at least one of its corresponding original alarms is also an error, and vice versa. In other words, when the repositioned alarms R are false positives, the original alarms Φ_S are also false positives, and vice versa. Thus, NCD-based repositioning with these two constraints, $C1$ and $C2$, meets the *repositioning criterion* (Section 1). As NCD-based repositioning creates new alarms, with the third constraint $C3$, we ensure that the repositioning never results in more alarms than the input for repositioning. Thus, NCD-based repositioning performed with constraints $C1$, $C2$, and $C3$ is safe, without spurious error detection by the repositioned alarms, and without increasing the overall number of alarms.

For example, Figure 1 also shows NCD-based repositioning of the similar alarms, obtained using the NCDs computed above (Section 4.2). The repositioned alarms are shown using green circles. The shown NCD-based repositioning satisfies the three repositioning constraints ($C1$, $C2$, and $C3$).

During the repositioning of a set of similar alarms, when a repositioned alarm can be created at multiple locations satisfying the three repositioning constraints, we choose the location that is closer to its corresponding original alarms. Note that, although NCD-based clustering is performed using approximated NCDs, the repositioning obtained *is still safe* (Constraint $C1$).

When the approximate NCDs computation results in identifying ICDs of a group of similar alarms as NCDs, the obtained repositioned alarm(s) may result in detection of a spurious error. Due to this, (1) educating the tool user about the spurious error detection is required; and (2) we also report traceability links between the repositioned and their corresponding original alarms. The traceability links help user to inspect the corresponding original alarms when when a repositioned alarm is found to be an error. We experimentally evaluate the spurious error detection rate incurred due to computing the NCDs approximately.

Moreover, when the approximate ICDs/NCDs computation results identifying NCDs of a group of similar alarms as ICDs, NCD-based repositioning fails to reposition those alarms.

5 NCD-based Repositioning Technique: Algorithm

This section presents a technique for NCD-based repositioning of alarms. The technique computes ICDs of the alarms instead of NCDs: ICDs and NCDs of an alarm are mutually exclusive. For efficiency the technique is designed to compute ICDs of alarms while the alarms are repositioned: we do not compute the ICDs separately before the repositioning is performed. We begin describing the technique by defining *live alarm conditions* similarly to the *live variables* [16].

Definition 2 (Live Alarm-condition). *An alarm condition c is said to be* live *at a program point p, if a path from p to the program exit contains an alarm ϕ reported at a program point q with c as its alarm condition, and the path segment from p to q is definition free for any operand of c.* □

For example, in Figure 1b, condition $ny \neq 0$ is live at $exit(n_{34})$ due to the alarms D_{42} or D_{48}. However, the same condition is not live at $entry(n_{33})$.

5.1 Live Alarm-conditions Analysis

Analysis Overview In this analysis, alarm conditions of *a given set of original alarms* Φ are propagated in the backward direction by computing them as live alarm-conditions (*liveConds*). We use data flow analysis [16, 28] to compute liveConds at every program point in the program. The aim of this analysis, that we call *liveConds analysis*, is to compute repositioned alarms for Φ. To this end, for every liveCond ℓ that we compute at a program point p, we also compute the following information.

1. The original alarm(s) due to which ℓ is a liveCond at p. We refer to these alarms as *related original alarms* (relOrigAlarms) of ℓ.
2. The program point(s) that are later used to create repositioned alarms: a (new) repositioned alarm with ℓ as its alarm condition is created at each of these program points. In other words, these program points denote the locations where the relOrigAlarms of ℓ are to be repositioned. Thus, we refer to these program points as *repositioning locations* (reposLocations) of ℓ. A reposLocation of ℓ is either the location of an original alarm due to which ℓ is a liveCond at p, or a program point computed during its backward propagation (the *meet operation* discussed later).
3. The transitive control dependencies of the reposLocations of ℓ such that for every dependency there exists a condition-wise equivalent dependency on all the paths from p to every reposLocation. We refer to these dependencies as *relatedICDs* of ℓ, because their conditions denote at least one safety condition of the alarms that will get created at the reposLocations of ℓ.

To compute traceability links between the repositioned alarms and their corresponding original alarms (and vice versa), we compute the relOrigAlarms of ℓ reposLocation-wise: reposLocations of ℓ are the program points where relOrigAlarms of ℓ are to be repositioned. We refer to the alarms computed corresponding to a reposLocation p as *relOrigAlarms* of p. The relOrigAlarms of ℓ can be obtained by collecting together the relOrigAlarms of reposLocations of ℓ.
Notations Let $\langle \mathcal{N}, \mathcal{E} \rangle$ be the control flow graph of the program: $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is the set of edges. Let $\mathcal{P}$ be the set of all program points in the program. Let $\mathcal{E}_c \subset \mathcal{E}$ be the set of all conditional edges in the CFG, i.e., the set of all transitive control dependencies of each $p \in \mathcal{P}$. Let $\mathcal{L}$ be the set of all alarm conditions of a given set of original alarms Φ. Thus, the liveConds computed by the liveConds analysis at a program point are given by a subset of $\mathcal{L}$.

For a liveCond ℓ computed at a program point p, the reposLocations of ℓ and their corresponding relOrigAlarms[3] are given by a subset of $2^{\mathcal{A}}$ where $\mathcal{A} = \mathcal{P} \times 2^{\Phi}$. Thus, the values computed for a liveCond ℓ—its reposLocations (with their corresponding relOrigAlarms) and its relatedICDs—are given by an element of X, where $\mathrm{X} = 2^{\mathcal{A}} \times 2^{\mathcal{E}_c}$. We use a function $\mathrm{f} : \mathcal{L} \rightarrow \mathrm{X}$ that maps a liveCond $\ell \in \mathcal{L}$ to a pair of its reposLocations $A \in 2^{\mathcal{A}}$ and relatedICDs

[3] Note that the related original alarms (relOrigAlarms) of a liveCond ℓ are computed corresponding to its reposLocations (reposLocation-wise).

Given $S, S' \in \mathcal{B}$:

$$S \ {}^{n}\sqcap_{\mathcal{B}} \ S' = \bigcup_{\ell \in \left(condsIn(S) \ \cup condsIn\ (S')\right)} \{\ meetInfo(\ell, n, S, S')\ \} \tag{1}$$

$$meetInfo(\ell, n, S, S') = \begin{cases} merge(\ell, n, A, E, A', E') & \langle \ell, A, E\rangle \in S,\ \langle \ell, A', E'\rangle \in S' \\ \langle \ell, A, E\rangle & \langle \ell, A, E\rangle \in S,\ \ell \notin condsIn(S') \\ \langle \ell, A', E'\rangle & \langle \ell, A', E'\rangle \in S',\ \ell \notin condsIn(S) \end{cases} \tag{2}$$

$$merge(\ell, n, A, E, A', E') = mergeInfo(\ell, n, A, A',\ meetICDsInfo(E, E')) \tag{3}$$

$$meetICDsInfo(E, E') = \left\{ \ e, e' \ \middle|\ \begin{array}{c} e \in E,\ \ e' \in E', \\ e \text{ and } e' \text{ are equivalent condition-wise} \end{array} \right\} \tag{4}$$

$$mergeInfo(\ell, n, A, A', E) = \begin{cases} \langle \ell, reposAlarm\ (n, A, A')\ , \emptyset\rangle & points(A) \neq points(A'),\ E = \emptyset \\ \langle \ell, A \cup A', E\rangle & \text{otherwise} \end{cases} \tag{5}$$

$$reposAlarm(n, A, A') = \{\ \langle entry(n), origAlarms(A) \cup origAlarms(A')\rangle\} \tag{6}$$

$E \in 2^{\mathcal{E}_c}$. We write the liveCond ℓ with the mapped values as tuple $\langle \ell, A, E\rangle$. Thus, at a program point p, the liveConds analysis computes a subset of $\mathcal{L}_b$, where $\mathcal{L}_b = \{\langle \ell, A, E\rangle \mid \ell \in \mathcal{L},\ \mathrm{f}(\ell) = \langle A, E\rangle\}$.

For a given set $S \subseteq \mathcal{L}_b$ and $A \in 2^{\mathcal{A}}$ we define:

- $condsIn(S) = \{\ell \mid \langle \ell, A', E'\rangle \in S\}$, the set of all liveConds in S.
- $points(A) = \{p \mid \langle p, \Phi'\rangle \in A\}$, the set of all reposLocations in A.
- $origAlarms(A) = \cup_{\langle p, \Phi'\rangle \in A}\ \Phi'$, the set of all relOrigAlarms in A.

Lattice of liveConds Analysis As liveConds analysis computes subsets of $\mathcal{L}_b$ flow-sensitively at every program point $p \in \mathcal{P}$, we denote the lattice of these values by $\langle \mathcal{B} = 2^{\mathcal{L}_b}, {}^{n}\sqcap_{\mathcal{B}}\rangle$. We use ${}^{n}\sqcap_{\mathcal{B}}$ to denote the *meet* of the values flowing in at the *exit* of a branching node n. For simplicity of the equation, we have assumed that the branching node n corresponding to a meet operation is known when the meet is performed. This meet operation is shown using Equation 1, and it is idempotent, commutative, and associative. The meet operation for a liveCond ℓ is described below.

1. When ℓ flows-in at the meet point through only one branch, its reposLocations and relatedICDs remain unchanged (Equation 2).
2. Following are the updates when (i) ℓ flows-in at the meet point through both the branches, (ii) the reposLocations of ℓ flowing in through both the branches are different; and (iii) the relatedICDs of ℓ flowing in through both the branches does not have a condition-wise equivalent dependency (Equations 2 and 5). The reposLocations of ℓ are updated to $entry(n)$, and the relOrigAlarms of this reposLocation are obtained by combining together all the relOrigAlarms of ℓ flowing in through both the branches. Moreover, the relatedICDs of ℓ are updated to $\emptyset$. These updates denote creation of a new reposLocation $entry(n)$: we use $entry(n)$ instead of the meet point $exit(n)$ assuming that the branching nodes do not update values of a variable.
3. In the cases other than (1) and (2), the reposLocations of ℓ flowing in from both the branches are combined together without updating their respective

Let $m, n \in \mathcal{N}$; $e \in \mathcal{E}$; $\phi, \phi' \in \Phi$; $\ell, \ell' \in \mathcal{L}$; $S \in B$.

$$Out_n = \begin{cases} \emptyset & n \text{ is } End \text{ node} \\ \prod^{n}_{m \in pred(n)} {}^{B} \; Edge_{e \equiv n \to m}(In_m) & \text{otherwise} \end{cases} \tag{7}$$

$$Edge_{e=n\to m}(S) = \{\langle \ell, A, E \cup handleCtrlDep(e, A)\rangle \mid \langle \ell, A, E\rangle \in S\} \tag{8}$$

$$handleCtrlDep(e, A) = \begin{cases} \{ e \} & e \text{ is a transitive control dependency of } p \in points(A) \\ \emptyset & \text{otherwise} \end{cases}$$

$$In_n = Gen_n(Survived_n) \cup (Survived_n \setminus GenRemoved(Survived_n)) \tag{9}$$

$$Survived_n = processForICDsKill(n, Out_n \setminus Kill_n(Out_n)) \tag{10}$$

$$Kill_n(S) = \left\{ \langle \ell, A, E\rangle \;\middle|\; \begin{array}{c} \langle \ell, A, E\rangle \in S, \; n \text{ contains a definition} \\ \text{of an operand of } \ell \end{array} \right\} \tag{11}$$

$$processForICDsKill(n, S) = \{\langle \ell, A, \; E \setminus killICDs(E, n)\rangle \mid \langle \ell, A, E\rangle \in S\} \tag{12}$$

$$killICDs(E, n) = \left\{ e \;\middle|\; \begin{array}{c} e \in E, \text{ and } n \text{ contains a definition} \\ \text{of an operand of } cond(e) \end{array} \right\} \tag{13}$$

$$Gen_n(S) = \{ \; createLiveCond(\phi, n, S) \mid n \text{ has alarm } \phi \in \Phi \text{ reported for it}\} \tag{14}$$

$$createLiveCond(\phi, n, S) = \begin{cases} createInfo(\phi, n, \{\phi\} \cup origAlarms(R)) & \langle \ell, R, C\rangle \in S, \\ & cond(\phi) = \ell \\ createInfo(\phi, n, \{\phi\}) & \text{otherwise} \end{cases} \tag{15}$$

$$createInfo(\phi, n, \Phi') = \langle cond(\phi), \{\langle entry(n), \Phi'\rangle\}, \emptyset\rangle$$

$$GenRemoved_n(S) = \left\{ \langle \ell, A, E\rangle \;\middle|\; \begin{array}{c} n \text{ has alarm } \phi \in \Phi \text{ reported for it}, \\ \langle \ell, A, E\rangle \in S, \; \ell = cond(\phi) \end{array} \right\} \tag{16}$$

Fig. 3: Data flow equations of the liveConds analysis.

relOrigAlarms, and the relatedICDs are updated to the control dependencies that are condition-wise equivalent (Equations 5 and 4).

Data Flow Equations Figure 3 shows data flow equations of the liveConds analysis that computes liveConds in intraprocedural setting. Out_n and In_n denote the values computed by the liveConds analysis, respectively, at the *exit* and *entry* of a node n (Equations 7 and 9, respectively).

Equation 14 indicates that a liveCond ℓ is generated for every original alarm ϕ reported for a node n, with $\emptyset$ as the relatedICDs of ℓ, and $entry(n)$ as the only reposLocation of ℓ. When the same liveCond l also flows in at $entry(n)$ from a successor of n, (i) the relOrigAlarms of the liveCond flowing in are also added to relOrigAlarms of the reposLocation $entry(n)$ (Equation 15); and (ii) propagation of the values of l flowing in at $entry(n)$ is stopped (Equation 16). With this computation and the meet operation (Equation 1), we ensure that at any program point there exists only one tuple for a liveCond and the values computed for it. Note that the reposLocations of a liveCond are updated only when the liveCond is generated (Equation 14) or the meet operation is performed (Equation 1).

Following are the updates to relatedICDs of a liveCond ℓ. (i) When ℓ gets propagated through a transitive control dependency e of its reposLocation, e is added to the relatedICDs of ℓ (Equation 8). (ii) For a relatedICD e of ℓ, if an

assignment node assigns values to a variable in $cond(\phi)$, then e is removed from the relatedICDs of ℓ (Equation 12).

For example, in Figure 1b, $nx \neq 0$ and $ny \neq 0$ are two liveConds computed by the liveConds analysis at entry(n_{34}), i.e. in In_{34}. At this point, the reposLocations (with their relOrigAlarms) and relatedICDs of the first liveCond, $nx \neq 0$, respectively are $\{\langle entry(n_{37}), \{D_{38}, D_{45}\}\rangle\}$ and $\emptyset$. Moreover, the reposLocations (with their relOrigAlarms) and relatedICDs of the second liveCond, $ny \neq 0$, respectively are $\{\langle entry(n_{41}), \{D_{42}, D_{48}\}\rangle\}$ and $\emptyset$.

Algorithm 1 Steps to perform NCD-based repositioning of alarms.

```
global R_Φ;
procedure performNCDRepositioning
    R_Φ = ∅;
    for each node n ∈ N do
        for each liveCond ℓ ∈ Kill_n(Out_n) do
            createReposAlarms(ℓ, Out_n);
        end for
    end for

    /* Special case for the liveConds reaching procedure entry */
    for each liveCond ℓ ∈ condsIn(In_Start) do
        createReposAlarms(ℓ, In_Start);
    end for

    /* Postprocessing of repositioned alarms */
    R'_Φ = performClustering(R_Φ);
    R_f = performFallBack(R'_Φ);

    return R_f; /* Set of final repositioned alarms */
end procedure

procedure createReposAlarms(ℓ, S)
    for each ⟨ℓ, A, E⟩ ∈ S do
        for each ⟨q, Φ'⟩ ∈ A do
            if ⟨ℓ', q⟩ ∈ R_Φ and ℓ ⇒ ℓ' then
                R_Φ = (R_Φ \ {⟨ℓ', q⟩}) ∪ {⟨ℓ, q⟩}; /* Creating new repositioned alarm */
                createLinks(⟨ℓ, q⟩, linksOf(⟨ℓ', q⟩) ∪ Φ');
            else if ⟨ℓ', q⟩ ∈ R_Φ and ℓ' ⇒ ℓ then
                createLinks(⟨ℓ', q⟩, Φ'); /* Create new traceability links only */
            else
                R_Φ = R_Φ ∪ {⟨ℓ, q⟩};    /* Creating new repositioned alarm */
                createLinks(⟨ℓ, q⟩, Φ');
            end if
        end for
    end for
end procedure
```

5.2 NCD-based Repositioning using liveConds Analysis Results

In Algorithm 1 we provide steps to perform NCD-based repositioning using results of the liveConds analysis described above (Section 5.1). The steps are discussed below.

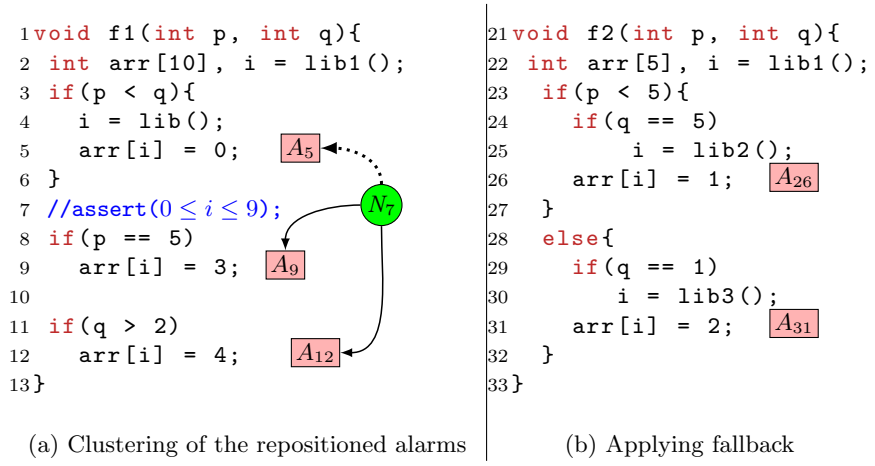
Creation of Repositioned Alarms As discussed in Section 5.1, the liveConds analysis results are used to create repositioned alarms for the original alarms Φ: the repositioned alarms are the results of NCD-based repositioning of Φ. For a liveCond ℓ computed at a program point p, a repositioned alarm $\langle \ell, q \rangle$ is created at each reposLocation q of ℓ (that is, ℓ is the condition of the alarm repositioned at every reposLocation of ℓ). Moreover, the relOrigAlarms of q are identified as the original alarms corresponding to the repositioned alarm $\langle \ell, q \rangle$, and thus use them to report the traceability links between the repositioned alarm $\langle \ell, q \rangle$ and its corresponding original alarms.

At every program point p, we collect the liveConds that are liveConds at p but not at a program point *just prior* to p, and use each of them to create repositioned alarms as described above. The liveConds to be collected are the liveConds that are killed at every node n, given by $Kill_n(Out_n)$. This approach to collect the liveConds removes redundancy in creating the repositioned alarms. As a special case, we collect the liveConds that reach the *procedure entry* (given by In_{Start}), because a liveCond can reach this point (Start node) when all the variables in the liveCond are *local and uninitialized*.

The above approach to collect the liveConds for creating the repositioned alarms ensures the following: each liveCond ℓ that got generated at p due to an original alarm $\phi_p \in \Phi$ gets collected and used to create a repositioned alarm along every path starting at the program entry and ending at p. Thus, along every path reaching p, there exists a repositioned alarm with $\ell = cond(\phi)$ as its alarm condition. As a consequence of this, the repositioned alarms corresponding to the original alarm ϕ_p together dominate ϕ_p. This indicates that the repositioning of Φ thus obtained is *safe*, i.e., the repositioning satisfies the constraint $C1$ (Section 4.3). Note that, the Equations 1, 8, and 12 together indicate that a repositioned alarm is created only when the constraint $C2$ is satisfied (Section 4.3).

Clustering of the Repositioned Alarms Let R_Φ be the set of all repositioned alarms created using the liveConds analysis results (described above). As a repositioned alarm can be a *dominant alarm* for another repositioned alarm, we postprocess R_Φ for their clustering using the state-of-the-art clustering techniques [20, 24, 34]. As an example, consider the code in Figure 4a that has three AIOB alarms reported at lines 5, 9, and 12. The repositioned alarms computed for these alarms are $R_\Phi = \{N_7, A_5\}$, where $N_7 = \langle 0 \leq i \leq 9, \text{entry}(n_8) \rangle$ with A_9 and A_{12} as its corresponding original alarms. Observe that N_7 is a dominant alarm of A_5. Thus, to further reduce the number of alarms, we cluster these two alarms. As a result, only one repositioned alarm N_7 gets reported with all the three original alarms as its corresponding original alarms.

Computation of Final Repositioned Alarms Let R'_Φ be the set of repositioned alarms obtained after their clustering discussed in Section 5.2. As a limitation of our technique, in rare cases, repositioning of a given set of original



(a) Clustering of the repositioned alarms

(b) Applying fallback

Fig. 4: Examples to illustrate postprocessing of the repositioned alarms.

alarms can result into more repositioned alarms than the original alarms. We illustrate this using the two AIOB alarms shown in Figure 4b. The repositioning of these two alarms results in three repositioned alarms $R_{\Phi} = \{A_{26}, A_{31}, \langle 0 \leq i \leq 9, \text{entry}(n_{23})\rangle\}$. This limitation case arises because variables in their conditions have different data dependencies. Clustering these repositioned alarms also results in the same set of the repositioned alarms, i.e., $R'_{\Phi} = R_{\Phi}$. Thus, we identify the cases where the repositioning of a group of similar alarms $\Phi' \subseteq \Phi$ results in more repositioned alarms than Φ'; and then apply fallback in these cases: we report Φ' instead of reporting their corresponding repositioned alarms. Thus, in this example, finally A_{26} and A_{31} get reported.

Note that the above limitation *may occur* only when the similar alarms being repositioned have different data dependencies. Avoiding such similar alarms in the input to NCD-based repositioning will miss to merge a few similar alarms, e.g., the similar alarms N_7 and A_5 discussed above. Thus, as we intend to reposition more similar alarms together, we accept all the tool-generated alarms as input to NCD-based repositioning and resort to fallback in such limitation cases. Additionally, we apply fallback when the repositioning of a group of similar alarms results in equal number of the repositioned alarms. Applying the fallback ensures that the repositioning obtained using the technique satisfies the constraint $C3$ (Section 4.3). Thus, our technique never increases the number of alarms reported to the user than the input original alarms.

5.3 Properties of the NCD-based Repositioning Technique

Theorem 1. *Given a set of alarms Φ, repositioning of Φ obtained using the NCD-based repositioning technique is* safe.

Proof. Let $\phi_p \in \Phi$ be an alarm. When the condition of the alarm, $cond(\phi_p)$ is generated as a liveCond at p, its reposLocation is p (Equation 14). This reposLo-

cation is updated only at the meet operation during the backward propagation of $cond(\phi_p)$, and only when the reposLocations flowing-in through both the branches are different (Equations 1). Thus, the reposLocations of the liveCond $cond(\phi_p)$ at a program point includes p or the meet points that *combinedly dominate* p (because the $cond(\phi_p)$ is computed as liveCond).

The reposLocations of $cond(\phi_p)$ at every program point where $cond(\phi_p)$ is killed, are used to create repositioned alrams: a repositioned alarm is created with $cond(\phi_p)$ as its alarm condition at each of its reposLocation. Collecting the liveConds for alarms repositioning this way ensures the following: (1) each liveCond ℓ generated at a program point p gets collected and used at least once along every path starting at program entry and ending at p, and (2) ℓ is repositioned at each of its reposLocation. This is sufficient to guarantee that for every originanl alarm ϕ_p there exists a repositioned alarm along every path reaching ϕ. Thus, repositioning performed is *safe*: all the behaviors of an original alarm ϕ_p are shown by its corresponding repositioned alarm(s). In other words, when ϕ_p is an error, at least one of its corresponding repositioned alarm is also an error. Thus, the repositioning obtained by the NCD-based repositioning technique is safe: detection of an error is not missed.

Moreover, the repositioning obtained after postprocessing of the repositioned alarms—for their clustering and applying fallback in the limitation cases—is still safe.

Theorem 2. *Given a set of alarms Φ, repositioning of Φ obtained using the NCD-based repositioning technique satisfies the three criteria of NCD-based repositioning.*

Proof. Constraint $C1$: Proof for satisfying this constraint by the repositioning obtained by the technique follows from Theorem 1.
Constraint $C2$: The Equations 1, 8, and 12 together indicate that a new reposLocation q is created at a meet point only when all the paths between q to its corresponding original alarms (relOrigAlarms) do not have ICDs of the alrams. A new repositioned alarm is created at this location q or at the program points of the original alrams. As the constraint $C2$ is satisfied in both the cases, the repositioning obtained using the technique also satisfies the constraint $C2$.
Constraint $C3$: Postprocessing the repositioned alarms for applying fallback in the cases that does not reduce alarms ensures that repositioning constraint $C3$ is satisfied.

6 Empirical Evaluation

In this section we evaluate NCD-based repositioning technique (Section 5) in terms of the reduction in the number of alarms.

Implementation We implemented NCD-based repositioning technique using analysis framework of our commercial static analysis tool, TCS ECA [32]. The analysis framework supports analysis of C and COBOL programs. The framework allows to implement data flow analyses using function summaries. We implemented the liveConds analysis to compute liveConds inter-functionally and

Table 2: Experimental results for NCD-based clustering.

(a) Open source applications

Application	Size (KLOC)	Input Alarms	% Reduction	Time (mins)	% Overhead
archimedes-0.7.0	0.8	2275	10.55	1.9	24.5
polymorph-0.4.0	1.3	25	12.00	0.6	27.5
acpid-1.0.8	1.7	25	8.00	0.4	23.5
spell-1.0	2.0	71	5.63	0.8	18.4
nlkain-1.3	2.5	319	1.57	0.5	15.7
stripcc-0.2.0	2.5	229	8.30	1.0	16.8
ncompress-4.2.4	3.8	92	3.26	0.5	23.6
barcode-0.96	4.2	1064	9.02	2.4	17.7
barcode-0.98	4.9	1310	9.08	2.8	15.7
combine-0.3.3	10.0	819	23.57	4.3	55.3
gnuchess-5.05	10.6	1783	15.09	8.6	95.4
antiword-0.37	27.1	613	9.95	26.7	72.2
sudo-1.8.6	32.1	7433	8.69	133.2	22.5
uucp-1.07	73.7	2068	6.58	21.6	7.5
ffmpeg-0.4.8	83.7	45137	10.41	239.0	11.6
sphinxbase-0.3	121.9	1516	5.67	6.5	17.3

(b) Industry apps. (C & COBOL)

Application	Size (KLOC)	Input Alarms	% Reduction	Time (mins)	% Overhead
C App 1	3.4	383	12.79	1.8	13.3
C App 2	14.6	422	2.37	4.5	15.8
C App 3	18.0	441	22.00	4.0	12.4
C App 4	18.1	1055	20.47	5.6	23.7
C App 5	18.3	535	23.55	4.7	12.5
C App 6	30.5	1001	29.77	5.1	23.4
C App 7	30.9	1379	17.19	42.3	2.8
C App 8	34.6	23404	4.28	186.9	17.8
C App 9	111.0	2241	12.72	7.0	22.2
C App 10	127.8	987	12.97	1.8	21.7
C App 11	187.2	4494	18.09	36.2	36.7
COBOL 1	11.4	341	5.57	1.1	78.3
COBOL 2	11.9	601	28.62	7.1	20.9
COBOL 3	16.7	499	0.40	6.4	179.4
COBOL 4	26.8	1158	32.21	25.7	63.0
COBOL 5	37.8	1826	36.09	3.7	80.0

by considering transivity. In the inter-functional implementation, the data flow analysis is solved in bottom-up order only: liveConds are propagated from a called-function to its callers but not from a caller-function to the called functions.

Selection of Applications and Alarms To evaluate the applicability and performance of NCD-based repositioning technique in different contexts, we select in total 32 applications that belonged to the following three categories. (i) 16 open source applications written in C and previously used as benchmarks for evaluating the alarms clustering and repositioning techniques [21, 27, 34]; (ii) 11 industry C applications from the automotive domain; and (iii) 5 industry COBOL applications from the banking domain.

We analyzed the applications using TCS ECA for five commonly checked categories of run-time errors (safety properties): *array index out of bounds* (AIOB), *division by zero* (DZ), *integer overflow underflow (OFUF)*, *uninitialized variables* (UIV), and *illegal dereference of a pointer* (IDP). The IDP property is not applicable for COBOL applications as COBOL programs do not have pointers. The tool-generated alarms are postprocessed using the alarms clustering techniques [21, 24] and then the resulting *dominant alarms* are postprocessed using the state-of-the-art repositioning [27]. The resulting repositioned alarms are provided as input to NCD-based repositioning. All the applications in the three sets were analyzed and the alarms were postprocessed using a machine with i7 2.5GHz processor and 16GB RAM.

Results Table 2 presents the evaluation results as per the categories of the applications (open source and industry). The column *Input Alarms* presents the number of alarms that were given as input to NCD-based repositioning tech-

nique, while the column *% Reduction* presents the percentage reduction achieved in the number of alarms by the technique. The evaluation results indicate that, compared to state-of-the-art repositioning, NCD-based repositioning technique reduces the number of alarms on the three sets of applications—open source, C industry, and COBOL industry—by up to 23.57%, 29.77%, and 36.09% respectively. The median reductions are 9.02%, 17.18%, and 28.61%, respectively. Moreover, the average reductions respectively are 10.16%, 8.97%, and 27.68%.

The column *Time* in Table 2 presents the time taken to (i) analyze the applications for those five properties, (ii) postprocess the TCS ECA-generated alarms using the clustering and the state-of-the-art repositioning techniques. The columns *% Overhead* presents the performance overhead incurred due to the extra time taken by NCD-based repositioning technique. We believe the performance overhead added is acceptable because the alarms reduction can be expected to reduce the users' manual effort which is much more expensive than machine time. Moreover, the reduced alarms may result in performance gain when the alarms are postprocessed for false positives elimination using time-expensive techniques like model checking.

Other observations:

(1) We measured the reduction in the number of alarms generated for each of the properties selected. The median reductions computed property-wise on all the applications, are 25.8% (AIOB), 45.72% (DZ), 6.89% (OFUF), 18.17% (UIV), and 10.3% (IDP). (2) The fallback got applied (Section 5.2) in 2592 instances during NCD-based repositioning of the total 105,546 alarms. (3) We measured distribution of similar alarms in alarms resulting after NCD-based repositioning. Results of this study are shown under *After NCD-based repositioning* in Table 3 (right side). For comparison purpose, we also show distibution of similar alarms in alarms resulting after state-of-the-art clustering and repositioning techniques, i.e., before NCD-based repositioning (left side). The results indicate that around 43% of the dominant alarms resulting after NCD-based repositioning on the open source applications are found to be similar alarms, and 64% of these similar alarms appear in the repositioning limitation scenarios. See in Table 3. Our manual analysis of 200 alarms appearing in these limitation scenarios showed they are not merged together due to (i) presence of common safety conditions (ICDs), (ii) limitations in our implementation to compute the liveConds inter-functionally, or (iii) the fallback got applied.

Evaluation of Spurious Error Detection by the Repositioned Alarms As discussed in Section 4.3, a repositioned alarm obtained through repositioning based on the approximated NCDs can be a spurious error. A repositioned alarm is a spurious error when a NCD computed with our approach is actually an ICD. To measure the spurious error detection rate, we manually analyzed 150 repositioned alarms that were created due to merging of two or more similar alarms: each repositioned alarm has two or more original alarms corresponding to it. The analyzed alarms are randomly selected from the repositioned alarms generated on the first nine open source applications (Table 2a) and two industry applications (C applications 4 and 7 in Table 2b). These selected 150 repositioned

Table 3: Distribution of similar alarms in alarms reported after state-of-the-art clustering and repositioning techniques (left side) and NCD-based repositioning (right side). The column *Total Alarms* shows the number of total alarms generated by the techniques for the selected five properties. The column *% Similar Alarms* presents percentage of similar alarms in the total alarms. The column *% Same DDs* (resp. *% Different DDs*) presents percentage of the similar alarms that have *same data dependencies* (resp. *different data dependencies*).

Application	After state-of-the-art clustering and repositioning				After NCD-based repositioning			
	Total Alarms	% Similar Alarms	% Same DDs	% Different DDs	Total Alarms	% Similar Alarms	% Same DDs	% Different DDs
archimedes-0.7.0	2275	51.60	34.24	65.76	2035	43.29	11.80	88.20
polymorph-0.4.0	25	28.00	100.00	0.00	22	13.64	100.00	0.00
acpid-1.0.8	25	44.00	45.45	54.55	23	39.13	22.22	77.78
spell-1.0	71	25.35	44.44	55.56	67	17.91	16.67	83.33
nlkain-1.3	319	53.92	36.63	63.37	314	52.87	34.34	65.66
stripcc-0.2.0	229	66.81	84.31	15.69	210	60.95	82.81	17.19
ncompress-4.2.4	92	51.09	53.19	46.81	89	48.31	48.84	51.16
barcode-0.96	1064	47.09	61.68	38.32	968	38.84	52.66	47.34
barcode-0.98	1310	46.64	60.72	39.28	1191	38.29	49.56	50.44
combine-0.3.3	819	66.42	71.14	28.86	626	47.76	44.82	55.18
gnuchess-5.05	1783	51.65	55.27	44.73	1514	40.55	31.76	68.24
antiword-0.37	613	32.79	60.70	39.30	552	26.45	45.89	54.11
sudo-1.8.6	7433	43.72	54.18	45.82	6787	35.72	39.44	60.56
uucp-1.07	2068	51.50	70.23	29.77	1932	45.91	63.59	36.41
ffmpeg-0.4.8	45137	51.99	82.33	17.67	40439	44.34	72.98	27.02
sphinxbase-0.3	1516	54.68	49.70	50.30	1430	50.42	38.70	61.30
Total	**64779**	**50.89**	**74.55**	**25.45**	**58199**	**43.12**	**63.76**	**36.24**

alarms have in total 482 original alarms corresponding to them. In our manual analysis, we checked each of the selected alarms whether it is a spurious error. We found three repositioned alarms to be spurious errors, and thus, the spurious error detection rate to be 2%. This indicates that our approach to compute NCDs/ICDs of similar alarms is effective, and for the analyzed cases, NCD-based repositioning technique reduced the number of alarms by 70% but at the cost of detecting a few spurious errors (2%).

7 Related Work

Heckman and Williams [14], and Muske and Serebrenik [26] have recently surveyed literature on postprocessing static analysis alarms. Among the techniques surveyed, our approach to reposition alarms belongs to the category of *clustering of alarms* together with the work of Lee et al. [20], Muske et al. [24, 27] and Zhang et al. [34]. However, those techniques are unable to group some of the

similar alarms which could be grouped/merged together (discussed in Section 1). Among those techniques, as the state-of-the-art repositioning technique [27] overcomes the limitations of the other alarms-clustering techniques [20, 24, 34], we compared and evaluated our NCD-based repositioning against it.

On the similar lines to alarms repositioning, Cousot et al. [8] have proposed hoisting necessary preconditions for providing the preconditions required by the Design by Contract [23]. Furthermore, Muske et al. [25] have proposed grouping the related/similar alarms based on similarity of modification points. In their approach [24], as the grouped alarms are inspected using values at the modification points of alarm variables, the inspection often finds spurious errors when the alarms are actually false positives solely due to their transitive control dependencies (ICDs). However, none of these techniques [8,24,25] identifies the conditional statements (control dependencies) that are non-impacting to the similar alarms.

Kumar et al. [18] identify the conditional statements that are value-impacting to the alarms. However, the notion of *value-impacting conditional statements* (resp. non value impacting conditional statements) is different from the ICDs (resp. NCDs) of the alarms. That is, a transitive control dependency identified as *non value-impacting* to an alarm can actually be an ICD of the alarm, and a control dependency identified as value-impacting can be an NCD. For example, in Figure 2, the control dependency $n_7 \rightarrow n_8$ of A_8 is ICD, whereas the technique by Kumar et al. identifies the same dependency is non-value impacting. To the best of our knowledge, no other static analysis technique or alarms postprocessing technique has formally proposed the notion of NCDs/ICDs of alarms or used them in alarms postprocessing.

As the NCD-based clustering of alarms is orthogonal to other alarms postprocessing techniques, it can be applied in conjunction with those. We believe that the combinations will provide more benefits as compared to the benefits obtained by applying them individually.

8 Conclusion

We have proposed the notion of NCDs of alarms, and NCD-based repositioning to reduce the number of alarms. Our approach to compute approximated NCDs of similar alarms is observation-based, and the computation is based on whether conditions in the enclosing conditional statements of a group of similar alarms are equivalent. This approximated approach is required, because the existing alarms clustering and repositioning techniques, being conservative, still report high percentage of similar alarms. The reported large number of alarms increases the cost to postprocess them manually or automatically.

We performed an evaluation of NCD-based repositioning using a large set of alarms on three kinds of applications, 16 open source C applications, 11 industry C applications, and 5 industry COBOL applications. The evaluation results indicate that, compared to the state-of-the-art repositioning technique, NCD-based repositioning reduces the number of alarms respectively by up to 23.57%, 29.77%, and 36.09%. The median reductions are 9.02%, 17.18%, and 28.61%,

respectively. Our manual analysis showed that our approach to approximately compute NCDs of similar alarms is effective: the approximation helped to reduce the alarms in the analyzed cases by 70%, however it resulted in 2% of the repositioned alarms detecting a spurious error.

We believe that NCD-based repositioning, being orthogonal to many of the existing approaches to postprocess alarms, can be applied in conjunction with those approaches. We plan to (i) explore a few more techniques to compute NCDs for alarms (similar as well as non-similar alarms); and (2) use the NCDs to improve the other alarms-postprocessing techniques like automated false positives elimination and version-aware static analysis.

References

1. Allen, F.E.: Control flow analysis. In: Symposium on Compiler Optimization. pp. 1–19. ACM, New York, NY, USA (1970)
2. Ayewah, N., Pugh, W.: The Google FindBugs fixit. In: International Symposium on Software Testing and Analysis. pp. 241–252. ACM, New York, NY, USA (2010)
3. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: Workshop on Program Analysis for Software Tools and Engineering. pp. 1–8. ACM, New York, NY, USA (2007)
4. Beller, M., Bholanath, R., McIntosh, S., Zaidman, A.: Analyzing the state of static analysis: A large-scale evaluation in open source software. In: International Conference on Software Analysis, Evolution, and Reengineering. vol. 1, pp. 470–481 (2016)
5. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (2010)
6. Brat, G., Venet, A.: Precise and scalable static program analysis of nasa flight software. In: 2005 IEEE Aerospace Conference. pp. 1–10 (March 2005)
7. Christakis, M., Bird, C.: What developers want and need from program analysis: An empirical study. In: International Conference on Automated Software Engineering. pp. 332–343. ACM, New York, NY, USA (2016)
8. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic Inference of Necessary Preconditions, pp. 128–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (Oct 1991)
10. Denney, E., Trac, S.: A software safety certification tool for automatically generated guidance, navigation and control code. In: 2008 IEEE Aerospace Conference. pp. 1–11 (March 2008)
11. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: Conference on Programming Language Design and Implementation. pp. 181–192. ACM, New York, NY, USA (2012)
12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (Jul 1987)
13. Gehrke, M.: Bidirectional Predicate Propagation in Frama-C and its Application to Warning Removal. Master's thesis, Hamburg University of Technology (2014)

14. Heckman, S., Williams, L.: A systematic literature review of actionable alert identification techniques for automated static code analysis. Inf. Softw. Technol. **53**(4), 363–387 (2011)
15. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: International Conference on Software Engineering. pp. 672–681. IEEE Press, Piscataway, NJ, USA (2013)
16. Khedker, U., Sanyal, A., Sathe, B.: Data flow analysis: theory and practice. CRC Press (2009)
17. Kornecki, A., Zalewski, J.: Certification of software for real-time safety-critical systems: state of the art. Innovations in Systems and Software Engineering **5**(2), 149–161 (Jun 2009)
18. Kumar, S., Sanyal, A., Khedker, U.P.: Value Slice: A New Slicing Concept for Scalable Property Checking, pp. 101–115. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
19. Layman, L., Williams, L., St. Amant, R.: Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In: International Symposium on Empirical Software Engineering and Measurement. pp. 176–185 (2007)
20. Lee, W., Lee, W., Kang, D., Heo, K., Oh, H., Yi, K.: Sound non-statistical clustering of static analysis alarms. ACM Trans. Program. Lang. Syst. **39**(4), 16:1–16:35 (Aug 2017)
21. Lee, W., Lee, W., Yi, K.: Sound non-statistical clustering of static analysis alarms. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 299–314. Springer-Verlag, Berlin, Heidelberg (2012)
22. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 462–473. ESEC/FSE 2015, ACM, New York, NY, USA (2015)
23. Meyer, B.: Design by contract. Prentice Hall (2002)
24. Muske, T., Baid, A., Sanas, T.: Review efforts reduction by partitioning of static analysis warnings. In: International Working Conference on Source Code Analysis and Manipulation. pp. 106–115 (2013)
25. Muske, T., Khedker, U.P.: Cause points analysis for effective handling of alarms. In: International Symposium on Software Reliability Engineering. pp. 173–184 (Oct 2016)
26. Muske, T., Serebrenik, A.: Survey of approaches for handling static analysis alarms. In: International Working Conference on Source Code Analysis and Manipulation. pp. 157–166 (2016)
27. Muske, T., Talluri, R., Serebrenik, A.: Repositioning of static analysis alarms. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 187–197. ISSTA 2018, ACM, New York, NY, USA (2018)
28. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
29. Rival, X.: Abstract dependences for alarm diagnosis. In: Yi, K. (ed.) Programming Languages and Systems. pp. 347–363. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
30. Rival, X.: Understanding the origin of alarms in astrÉe. In: Proceedings of the 12th International Conference on Static Analysis. pp. 303–319. SAS'05, Springer-Verlag, Berlin, Heidelberg (2005)
31. Sadowski, C., van Gogh, J., Jaspan, C., Söderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: International Conference on Software Engineering. pp. 598–608. IEEE Press, Piscataway, NJ, USA (2015)

32. TCS Embedded Code Analyzer (TCS ECA): `https://www.tcs.com/tcs-embedded-code-analyzer`, [Online accessed 30-Aug-2019]
33. Venet, A.: A practical approach to formal software verification by static analysis. Ada Lett. **XXVIII**(1), 92–95 (2008)
34. Zhang, D., Jin, D., Gong, Y., Zhang, H.: Diagnosis-oriented alarm correlations. In: Asia-Pacific Software Engineering Conference. vol. 1, pp. 172–179 (2013)