Classification and Ranking of Delta Static Analysis Alarms

Tukaram Muske
Tata Consultancy Services Ltd
Pune, India
tukarammuske@gmail.com

Alexander Serebrenik
Eindhoven University of Technology
Eindhoven, The Netherlands
a.serebrenik@tue.nl

Abstract—Static analysis tools help to detect common programming errors but generate a large number of false positives. Moreover, when applied to evolving software systems, around 95% of alarms generated on a version are repeated, i.e., they have also been generated on the previous version. Version-aware static analysis techniques (VSATs) have been proposed to suppress the repeated alarms that are not impacted by the code changes between the two versions. The alarms reported by VSATs after the suppression, called delta alarms, still constitute 63% of the tool-generated alarms.

We observe that delta alarms can be further postprocessed using their corresponding code changes: the code changes due to which VSATs identify them as delta alarms. However, none of the existing VSATs or alarms postprocessing techniques postprocesses delta alarms using the corresponding code changes. Based on this observation, we use the code changes to classify delta alarms into six classes that have different priorities assigned to them. The assignment of priorities is based on the type of code changes and their likelihood of actually impacting the delta alarms. The ranking of alarms, obtained by prioritizing the classes, can help suppress alarms that are ranked lower, when resources to inspect all the tool-generated alarms are limited.

We performed an empirical evaluation using 9789 alarms generated on 59 versions of seven open source C applications. The results indicate that the proposed classification and ranking of delta alarms help to identify, on average, 53% of delta alarms as more likely to be false positives than the others.

I. INTRODUCTION

Static analysis tools have shown promise in detection of common programming errors in software systems and also proving absence of those defects [1]–[5]. Despite this, recent studies [6]–[8] report that these tools are underused in practice. The studies report that the large number of alarms generated and the effort required to manually partition them into true positives and false positives are two primary reasons for the underuse. The manual inspection of alarms for the partitioning is found to be time-consuming and error-prone [6], [9], [10].

Numerous alarms generated on a version of evolving software system are repeated, i.e., they have also been generated on the previous version of the system. A few of the alarms postprocessing techniques [11] propose to reduce the number of alarms by suppressing repeated alarms that are not impacted by the code changes between the two versions [12]–[14] (see Section II-B). We call these postprocessing techniques version-aware static analysis techniques (VSATs) and the alarms reported by them delta alarms.

Due to their limitations, VSATs still report 40-80% of alarms generated by static analysis tools as delta alarms [12], [13]. Our pilot study (see Section III) also indicated that around 63% of alarms generated by static analysis tools (tool-generated alarms) get reported as delta alarms. That is, the number of delta alarms is still large, and the alarms need to be processed to reduce their number and simplify their manual inspection.

We find that, in addition to computing delta alarms, code changes between two consecutive versions can be used further to postprocess the alarms. However, none of the VSATs or existing alarms postprocessing techniques postprocesses delta alarms based on the code changes. Based on this observation, to address the problem of large number of delta alarms, we propose to postprocess the alarms by taking into account their corresponding code changes: the code changes due to which VSATs report them as delta alarms.

In our proposed technique, we classify delta alarms into six classes depending on type of their corresponding code changes. These six classes have different priorities assigned to them. The assignment of priorities is motivated by our observation that different types of program statements impact alarms differently, and therefore, changes made to those impacting program statements will impact the alarms differently.

The prioritization of classes allows to rank alarms. Since the alarms in the lowest priority class(es) are more likely to be false positives, they can be suppressed. The alarms suppression may result in suppressing a true positive, however it is unavoidable when the resources available to manually inspect all the tool-generated alarms are not sufficient. The proposed postprocessing of delta alarms is orthogonal to techniques that are proposed for postprocessing of alarms [11], [15], [16]. The proposed technique is more suitable to alarms generated by deep static analysis tools, i.e., tools that analyse flow of data, (e.g., Astree [17]), than to alarms that are generated by a tool based on pattern-matching (e.g., FindBugs [18]).

We performed an empirical evaluation of the proposed technique using 9789 alarms generated by a commercial static analysis tool on 59 versions of seven open source C applications. The results indicate that the proposed classification and ranking of delta alarms help to identify, on average, 53% of delta alarms as more likely to be false positives than the others.
The key contribution of the paper is a novel technique that ranks delta alarms by classifying them based on their corresponding code changes.

Paper Outline: Section II briefly describes motivation to classify and rank delta alarms. Section III presents a pilot study that we conducted. Section IV describes the terms and notations used in the paper. Sections V and VI respectively describe the proposed classification and prioritization of delta alarms. Section VII discusses our empirical evaluation. Section VIII presents related work, and Section IX concludes.

II. Motivation

In this section, we briefly describe VSATs, their limitations, and motivation to classify and rank delta alarms. We begin by presenting a running example used throughout the paper.

A. Running Example

Consider the C code example in Figure 1 showing two consecutive versions $V_1$ and $V_2$. The code is simplified considerably but it is still sufficiently rich to motivate and present the proposed classification and ranking of delta alarms. We assume that all program points are reachable. The code changes between the two versions are typeset on grey background.

Analysis of $V_1$ (resp. $V_2$), using a static analysis tool for division by zero and array index out of bounds (AIOB) verification properties will result in four (resp. six) alarms. We use $D_n$ and $A_n$ to respectively denote an alarm generated at line $n$ for these two properties. Henceforth, we use the notations $V_1$ and $V_2$ to denote two consecutive code versions.

B. Background: VSATs and Their Limitations

1) **VSATs:** The approaches used by VSATs $[12]$–$[14]$, $[19]$–$[21]$ to suppress repeated alarms vary greatly. The techniques that are based on syntactic location matching $[19]$ and coding patterns can result in a false negative. Therefore, we call them unsound VSATs and exclude them from the subsequent discussion. The other VSATs $[12]$–$[14]$, $[21]$ perform safe suppression of repeated alarms, i.e., they do not result in a false negative. We call these techniques sound VSATs. The safe suppression of alarms by sound VSATs is based on assumption that the user has inspected all alarms reported on the previous version and has taken corrective actions for the alarms that were identified as true positives.

The VSAT proposed by Chimdyalwar and Kumar $[13]$ performs impact analysis for each repeated alarm to determine whether the alarm is impacted by the code changes. It then suppresses the repeated alarms that are not impacted. The VSAT proposed by Logozzo et al. $[12]$, called verification modulo versions (VMV), first extracts semantic environment conditions from $V_1$, instruments the code in $V_2$, and then verifies the instrumented code. The VSAT proposed by Jana et al. $[21]$ uses a change-based alarm reporting approach that reports an alarm only if the alarm point lies on a newly introduced, potentially unsafe, execution path. We exclude differential assertion checking-based VSAT $[14]$ from our discussion, because it is not applicable to alarms that are newly generated, and non-scalability of the program verifiers on large programs is a concern.

2) **Limitations of VSATs:** The three sound VSATs described above have fundamentally different strengths and limitations. Due to their different strengths and limitations, they can be combined: an alarm is suppressed if any of them suppresses it. However, even their combination can fail to suppress delta alarms in commonly occurring scenarios. For example, none of these three VSATs or their combination suppresses any of the six alarms generated on $V_2$ shown in Figure 1. Thus, all those six alarms get reported as delta alarms.

Due to the conservative approaches used by sound VSATs, a high percentage of impacted alarms are spuriously generated (Section VI-C). For example, consider repeated alarm $A_{20}$. This alarm gets identified as impacted due to the change made on line 7. However, this change to values assigned to $x$ actually does not affect determining whether $A_{20}$ is a true positive: the values of $x$ do not restrict values taken by $y$ at line 25 but only control reachability of the alarm’s program point $[22]$.

Additionally, sound VSATs report a spurious delta alarm when the code from the two consecutive versions cannot be mapped precisely due to semantics preserving changes like code movement and refactoring. As a result, the number of delta alarms reported is still large: VSATs report around 40-80% of tool-generated alarms as delta alarms.

---

**Fig. 1:** Examples of delta alarms (yellow rectangles) and the classes identified for them (shown in comments).
C. Motivation for Classification/Ranking

The existing VSATs ([12], [13], [19], [20]) broadly classify delta alarms only into two classes: newly generated (the ones which did not occur on the previous version) and impacted (the repeated alarms that are impacted by the code changes). Among the six delta alarms shown, $A_{22}$, $D_{25}$, and $A_{27}$ are newly generated, and $A_{22}$, $A_{25}$, and $D_{25}$ are impacted.

Since the number of delta alarms is still large, their post-processing is required to reduce their number and simplify their manual inspection. We believe that classifying delta alarms further based on the type of their corresponding code changes is a more natural way to classify delta alarms as it captures the causal relationship, and expect that such a classification can provide multiple benefits.

In general, changes made between two consecutive versions are on different types of program statements, so are the delta alarms reported due to these changes. E.g., the reasons for generation of newly generated alarms $D_{25}$ and $A_{27}$ are different. The line 13 on which $D_{25}$ is reported, is newly added in $V_2$, whereas the array expression $a[i]$ checked by $A_{27}$ also existed in $V_1$ but was identified as safe. The change on line 13 generates $A_{27}$ for the same expression $a[i]$ in $V_2$.

As another example, consider impacted alarms $A_{22}$ and $A_{25}$ having their corresponding changes on lines 21 and 7 respectively. The changed program statements affect the two alarms differently: the change on line 21 directly modifies values of the index expression in $A_{22}$ whereas the change on line 7 only affects whether the program point of $A_{25}$ is reachable. In Section V-B we classify $A_{22}$ and $A_{25}$ (also $D_{25}$ and $A_{27}$) into different classes, based on the impact of their corresponding changes on them.

The classification of delta alarms allows to inspect the alarms differently depending on their class and simplify their manual inspection. For example, inspection of a newly generated alarm whose program statement is newly added will require inspecting the code on the backward slice generated for the alarm. However, for newly generated alarms whose POIs are converted from safe points in $V_1$ to alarms in $V_2$, inspecting only the corresponding code change(s) is sufficient.

Moreover, the classification can help us identify classes that are more important than the others. For example, based on impact of the corresponding code changes on $A_{27}$ and $D_{13}$ we can assign higher priority to the class of $A_{27}$ than the class of $D_{13}$ although $A_{27}$ and $D_{13}$ are newly generated alarms, the reasons for their generation are different and $A_{27}$ could be due to the side-effect of the code changes made between the versions. Similarly we can assign higher priority to the class of $A_{22}$ than the class of $A_{25}$ and thus rank $A_{22}$ before $A_{25}$.

III. PILOT STUDY

As discussed earlier (Section II-B), VSATs broadly classify delta alarms into two classes: newly generated and impacted. The usefulness of the classification and ranking technique we will develop is based on two assumptions: (1) the number of delta alarms reported by VSATs is large, and (2) a large percentage of delta alarms are impacted alarms. Indeed, if only few (impacted) delta alarms are reported, impact of the technique to be developed will be negligible. Hence, in this section we perform a preliminary study to measure (a) what percentage of alarms generated by static analysis tools (tool-generated alarms) are repeated; (b) what percentage of tool-generated alarms get reported as delta alarms; and (c) what percentage of delta alarms are newly generated and impacted alarms. The impacted alarms in the lowest priority classes, as described next in Section VI, are candidates for suppression by our technique.

We randomly chose four open source C applications from the list of 100 applications used by Cha et al. [23], with the constraints that (1) application size should be greater than 10 KLOC and less than 20 KLOC and (2) at least two versions of the application should be available online. Table I lists these four applications together with the number of versions selected, and the first and last versions in our selection.

We analyzed the selected 32 versions (Table I) for AIOB property using a commercial static analysis tool, TCS ECA [24]. We implemented impact analysis-based VSAT [15], and used the implementation to compute delta alarms from the tool-generated alarms. We measured the number of tool-generated alarms, repeated alarms, suppressed repeated alarms, delta alarms, newly generated alarms, and impacted alarms. The measurement results, shown in Table I indicate that, on average, (a) 95% of tool-generated alarms are repeated; (b) 63.3% of tool-generated alarms get reported as delta alarms and the other (36.7%) alarms get suppressed as they are repeated and not impacted by the code changes; and (c) 92% % (resp. 8%) of delta alarms are impacted (resp. newly

<table>
<thead>
<tr>
<th>Application</th>
<th>Versions selected</th>
<th>Tool-generated alarms</th>
<th>Repeated alarms</th>
<th>Suppressed repeated alarms</th>
<th>Impacted alarms</th>
<th>Newly generated alarms</th>
<th>Delta alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>7546</td>
<td>7170</td>
<td>2773</td>
<td>4397</td>
<td>376</td>
<td>4773</td>
<td></td>
</tr>
<tr>
<td>Percentage of the tool-generated alarms</td>
<td>95.0</td>
<td>36.7</td>
<td>58.2</td>
<td>4.9</td>
<td>63.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1To limit the amount of analysis and delta alarms’ computation time, we restricted the evaluation to relatively small applications.
generated). Therefore, we expect the proposed classification and ranking of delta alarms will be applicable to a large number of alarms and hence beneficial to simplify the manual inspection of alarms.

IV. TERMS AND NOTATIONS

A. Control Flow Graph

A control flow graph (CFG) \[25\] of a program is a directed graph \((\mathcal{N}, \mathcal{E})\), where \(\mathcal{N}\) is a set of nodes representing the program statements (including assignments and controlling conditions); and \(\mathcal{E}\) is a set of edges where an edge \(n \rightarrow n'\) represents a possible flow of program control from \(n \in \mathcal{N}\) to \(n' \in \mathcal{N}\). Depending on whether the program control flows conditionally or unconditionally along an edge, the edge is classified as conditional or unconditional respectively. For a conditional edge \(n \rightarrow n'\), we use label\((n \rightarrow n')\) to denote its label, and use condExpr\((n)\) to denote the conditional expression associated with the branching node \(n\). When a conditional edge \(n \rightarrow n'\) is from a switch statement to one of its case statements, we assume that the label of that edge is same as the case label. Since Figure 1 shows only one statement per line, we use \(n_m\) to denote the node of the program statement at line \(m\). We call the entry or exit of a node a program point.

B. Data and Control Dependencies

1) Data Dependencies: We call a node definition node (or assignment node) if it defines a variable. A variable \(x\) at a program point \(p\) is said to be data dependent on a definition node \(d_x\) if \(x = e\) is a reaching definition \[26\] of \(x\) at \(p\). Data dependencies of a variable \(v\) are the definitions on which \(v\) is data dependent. For an assignment node \(d_x\) : \(x = e\) of \(x\) (i.e., a data dependency of \(x\)), we use assignExpr\((d_x)\) to denote the assignment expression \(x = e\). We say that data dependencies of an assignment node (statement) \(d_x\) : \(x = e\) are same as union of data dependencies of variables in \(e\). For an expression other than assignment, its data dependencies are defined as the union of data dependencies of variables in it.

2) Control Dependencies: A node \(V\) is post-dominated by a node \(W\) if every directed path from \(V\) to the exit node (not including \(V\)) contains \(W\) \[27\]. Let \(X\) and \(Y\) be nodes in a control flow graph \(G\). \(Y\) is control dependent on \(X\) iff (1) there exists a directed path \(P\) from \(X\) to \(Y\) with any \(Z\) in \(P\) (excluding \(X\) and \(Y\)) post-dominated by \(X\) and (2) \(X\) is not post-dominated by \(Y\) \[27\]. Control dependencies of a node \(n\) are the conditional edges on which \(n\) is control dependent.

3) Transitive Data and Control Dependencies: Let \(C\) and \(D\) be the set of all possible conditional edges and assignment nodes in the program respectively. An assignment node \(d_x\) is called transitive data dependency of a variable \(x\) if \(d_x\) belongs to the transitive closure of data dependencies of \(x\). We use \(\alpha\) to denote a variable or expression at a program point, or a conditional edge. Let \(d \xrightarrow{\text{data}} \alpha\) denotes that \(d\) is a transitive data dependency of \(\alpha\). We denote transitive closure of data dependencies of \(\alpha\) using \(\text{dDep}^+ (\alpha)\), i.e., \(\text{dDep}^+ (\alpha) = \{d \mid d \in D, d \xrightarrow{\text{data}} \alpha\}\).

A conditional edge \(e\) is called transitive control dependency of \(\alpha\) if \(e\) belongs to the transitive closure of control dependencies of \(\alpha\). We write \(\text{cdDep} \xrightarrow{\text{control}} \alpha\) to denote \(\text{cdDep}\) is a data or control dependency of \(\alpha\). A definition or a conditional edge \(\text{cdDep}\) is a transitive data and control dependency of \(\alpha\), shown as \(\text{cdDep} \xrightarrow{\text{cdDep}} \alpha\), iff \(\text{cdDep}_1 \xrightarrow{\text{cdDep}} \text{cdDep}_2 \xrightarrow{\text{cdDep}} \text{cdDep}_3 \xrightarrow{\text{cdDep}} ... \xrightarrow{\text{cdDep}} \text{cdDep}_k\), where \(\text{cdDep}_1 = \text{cdDep}, \text{cdDep}_k = \alpha, \text{cdDep}_{i+1} \xrightarrow{\text{cdDep}} \text{cdDep}_{i+1}\), and \(k \geq 2\). We denote transitive closure of data and control dependencies of \(\alpha\) using \(\text{cdDep}^+ (\alpha)\), i.e., \(\text{cdDep}^+ (\alpha) = \{x \mid x \in C \cup D, x \xrightarrow{\text{cdDep}} \alpha\}\). Henceforth, we use dependency of \(\alpha\) to commonly refer to a data or control dependency that belongs to \(\text{cdDep}^+ (\alpha)\) (or \(\text{dDep}^+ (\alpha)\)). That is, a dependency is a definition or conditional expression.

C. Program Slicing

For a given program and a set of variable(s) at a program point of interest, program slicing \[22\], \[28\] computes a program that contains only those statements that are likely to influence values of the variables at that program point. Depending on use of program slices, several backward slicing techniques have been proposed \[29\], \[30\], such as backward slice \[28\], and thin slice \[31\]. Backward slice (resp. thin slice) generated for \(\alpha\) consists of program statements that correspond to dependencies in \(\text{cdDep}^+ (\alpha)\) (resp. \(\text{dDep}^+ (\alpha)\)).

D. Value Slice and Value Dependencies

Kumar et al. \[22\] have proposed the notion of value slice, which is a pruned version of backward slice and an enriched version of thin slice. A value slice generated for an expression \(e\), in addition to the transitive data dependencies of \(e\), also consists of the control dependencies that influence values of variables in \(e\). We call those control dependencies value dependencies. In other words, a value slice is obtained by eliminating from backward slice the control dependencies and their transitive dependencies, that only decide whether the program point of \(e\) is reachable. For example, the conditional edge \(n_{21} \rightarrow n_{25}\) is a value dependency of \(t\) present on line \(29\) because it controls assignment of values to \(t\). However, the same dependency is not a value dependency of expression \(b[y]\) present on line \(25\).

We use \(\text{vdDep} \xrightarrow{\text{value}} \alpha\) to denote \(\text{vdDep}\) is a data or value dependency of \(\alpha\). A definition or a conditional edge \(\text{vdDep}\) is a transitive data and value dependency of \(\alpha\) iff \(\text{vdDep}_1 \xrightarrow{\text{vdDep}} \text{vdDep}_2 \xrightarrow{\text{vdDep}} \text{vdDep}_3 \xrightarrow{\text{vdDep}} ... \xrightarrow{\text{vdDep}} \text{vdDep}_k\), where \(\text{vdDep}_1 = \text{vdDep}, \text{vdDep}_k = \alpha, \text{vdDep}_{i+1} \xrightarrow{\text{vdDep}} \text{vdDep}_{i+1}\), and \(k \geq 2\). We write \(\text{vdDep} \xrightarrow{\text{vdDep}} \alpha\) to denote that \(\text{vdDep}\) is a transitive data and value dependency of \(\alpha\). We denote transitive closure of data and value dependencies of \(\alpha\) using \(\text{vdDep}^+ (\alpha)\), where \(\text{vdDep}^+ (\alpha) = \{x \mid x \in C \cup D, x \xrightarrow{\text{vdDep}} \alpha\}\). Thus, the dependencies in \(\text{vdDep}^+ (\alpha)\) correspond to the program statements which appear in the value slice generated for \(\alpha\). Note that, for any expression \(\alpha\), \(\text{dDep}^+ (\alpha) \subseteq \text{vdDep}^+ (\alpha) \subseteq \text{cdDep}^+ (\alpha)\).

E. Static Analysis Alarms

We call an expression that is checked by a static analysis tool a point of interest (POI). For example, a POI for check
related to division by zero (DZ) property corresponds to a
denominator property. Let \( \text{poi}(\phi) \) denotes the POI of an
alarm \( \phi \). We use \( \phi_{1,2}^V \) to denote an alarm of a verification
property \( p \) and generated for a POI on line \( l \) in version \( V \).
We say that a slice generated for an alarm \( \phi \) is same as
the slice generated for \( \text{poi}(\phi) \). For an AIOB alarm having an
array access \( \text{arr}[i] \), including only the declaration of \( \text{arr} \) and
program statements corresponding to \( \text{cdDep}^+(i) \) in the slice
generated for the alarm is sufficient. We call two alarms of
same property similar iff their POIs are same.

We assume that a static analysis tool groups the generated
alarms using state-of-the-art clustering techniques \[32\], \[33\],
and a VSAT computes delta alarms from dominant alarms
resulting after the clustering. As a result, no two delta alarms
reported for a line are similar.

\section*{F. Code Mapping}

We create a mapping of the code from \( V_1 \) to \( V_2 \) using a
code mapping technique \[34\], \[35\]. We denote the map created
using Map\(_{1,2}^{V_1,V_2} \): \( \text{lines}(V_1) \rightarrow \text{lines}(V_2) \cup \{\bot\} \) which maps
source code lines in \( V_1 \) to their corresponding lines in \( V_2 \) and
to \( \bot \) if the lines are deleted from \( V_1 \). No two lines in \( V_1 \) map
to same line in \( V_2 \). We use this map to compute the following.

1) A line \( l_1 \) in \( V_1 \) is deleted iff Map\(_{1,2}^{V_1,V_2}(l_1) = \bot \).

2) A line \( l_2 \) is added in \( V_2 \) iff there does not exist \( l_1 \) in \( V_1 \)
such that Map\(_{1,2}^{V_1,V_2}(l_1) = l_2 \).

3) A line \( l_1 \) in \( V_1 \) or \( l_2 \) in \( V_2 \) is changed (resp. unchanged)
if Map\(_{1,2}^{V_1,V_2}(l_1) = l_2 \) and the code on \( l_1 \) and \( l_2 \), excluding
the white spaces, is different (resp. same).

When Map\(_{1,2}^{V_1,V_2}(l_1) = l_2 \) and \( l_2 \neq \bot \), we say that \( l_1 \) and \( l_2 \) are
corresponding lines. For a changed line \( l_1 \) in \( V_1 \) and its
corresponding line \( l_2 \) in \( V_2 \), similarly to the mapping of lines
in \( V_1 \) to \( V_2 \), we map every token (such as identifier, operator,
grouping symbol, or data type) in line \( l_1 \) to its corresponding
token in \( l_2 \) or to \( \bot \) if the token has been deleted from \( l_1 \).
Similarly to the lines mapping, the tokens mapping has one-to-
one correspondence, except when the tokens in \( l_1 \) of \( V_1 \) are
deleted or the tokens in \( l_2 \) of \( V_2 \) are added. We use Map\(_{1,2}^{V_1,V_2} \):
tokens\(_{1}(l_1) \rightarrow \) tokens\(_{2}(l_2) \cup \{\bot\} \) to denote the mapping
of tokens in \( l_1 \) to their corresponding tokens in \( l_2 \). Similarly
to determining if a line in \( V_1 \) (resp. \( V_2 \)) is deleted (resp. added),
changed, or unchanged discussed above, we use the mapping
of tokens to determine whether a given token in \( l_1 \) (resp. \( l_2 \))
is deleted (resp. added), changed, or unchanged.

Using the mapping of lines, i.e., Map\(_{1,2}^{V_1,V_2} \), and the mapping
of tokens in changed lines, we compute the following.

1) An expression \( e_1 \) at line \( l_1 \) in \( V_1 \) is deleted if (a) \( l_1 \) is
deleted from \( V_1 \), or (b) \( l_1 \) is changed and every token in
\( e_1 \) is deleted from \( l_1 \).

2) An expression \( e_2 \) is added to line \( l_2 \) in \( V_2 \) if (a) \( l_2 \) is
added to \( V_2 \), or (b) \( l_2 \) is changed and every token in \( e_2 \)
is added to \( l_2 \).

3) An expression \( e_1 \) at line \( l_1 \) in \( V_1 \) (resp. \( e_2 \) at line \( l_2 \) in
\( V_2 \)) is changed if at least one of the tokens in \( e_1 \) (resp.
\( e_2 \)) is changed.

4) An expression \( e_1 \) at line \( l_1 \) in \( V_1 \) is unchanged if (a) \( l_1 \)
is unchanged, or (b) \( l_1 \) is changed but none of the tokens in
\( e_1 \) is changed or deleted.

5) An expression \( e_2 \) at line \( l_2 \) in \( V_2 \) is unchanged if (a) \( l_2 \)
is unchanged, or (b) \( l_2 \) is changed but none of the tokens in
\( e_2 \) is changed or added.

We say that an expression \( e_1 \) at line \( l_1 \) in \( V_1 \) and an
expression \( e_2 \) at line \( l_2 \) in \( V_2 \) are corresponding expressions,
if (1) \( l_1 \) and \( l_2 \) are the corresponding lines, and (2) \( e_2 \) is a
changed version of \( e_1 \) or is same as \( e_1 \). We use the tokens-
based approach to determine if an expression that spans over
multiple lines is added, deleted, or changed, by matching
its sub-expressions appearing on different lines. To avoid
identifying semantically equivalent statements like \( i = i + 1 \)
and \( i++ \) as changed, we assume that the code has been
normalized \[36\]. Moreover, we assume that on each line, there
exists at most one program statement or a part of it.

\section{V. CLASSIFICATION OF ALARMS}

This section describes the proposed classification of delta
alarms.

\subsection*{A. Classification of Newly Generated Alarms}

We classify newly generated alarms into the below defined
classes: result-changed, POI-changed, and POI-added.

\textbf{Definition V.1} (Result-Changed Alarm). We call a newly
generated alarm \( \phi_{1,2}^{V_1,V_2} \) a result-changed alarm if its POI is
unchanged and no alarm of the property \( p \) was reported for
the POI’s corresponding expression in \( V_1 \).

In other words, for a result-changed alarm, its POI also
exists in \( V_1 \) and the tool’s analysis result for the POI is changed
from safe on \( V_1 \) to an alarm on \( V_2 \). For example, \( A_{27} \) is a
result-changed alarm.

\textbf{Definition V.2} (POI-Changed Alarm). We call a newly gener-
ated alarm \( \phi_{1,2}^{V_1,V_2} \) a POI-changed alarm if its POI is changed,
and an alarm of the property \( p \) was reported for the POI’s
corresponding expression in \( V_1 \).

For example, \( L_{22} \) is a POI-changed alarm, because its POI
\( y = 2 \) is changed from \( y \) in \( V_1 \), and an alarm of the same
property (DZ) was reported for the corresponding expression
\( y \) in \( V_1 \).

\textbf{Definition V.3} (POI-Added Alarm). We call a newly gener-
ated alarm \( \phi_{1,2}^{V_1,V_2} \) a POI-added alarm if its POI is added in \( V_2 \), or
its POI is changed and an alarm of the property \( p \) was not
reported for the POI’s corresponding expression in \( V_1 \).

For example, \( L_{13} \) is a POI-added alarm, because its line \( 13 \)
is added in \( V_2 \).

\subsection*{B. Classification of Impacted Alarms}

In this section, we present classification of impacted alarms.
Recall that, for an expression \( \alpha \), (a) \( \text{dDep}^+(\alpha) \) denotes
transitive closure of data dependencies of \( \alpha \); (b) \( \text{vdDep}^+(\alpha) \)
denotes transitive closure of data and value dependencies of
α; and (c) cdDep⁺(α) denotes transitive closure of data and control dependencies of α.

**Definition V.4 (Modified Dependency).** We call a dependency d of an expression α in V₁ (resp. V₂) a modified dependency if one of the following holds.

1) If d is an assignment node, assignExpr(d) is changed or deleted (resp. added).

2) If d is a conditional edge denoted as n → n’, label(d) is changed or condExpr(n) is changed or deleted (resp. added).

We first define impacted alarm.

**Definition V.5 (Impacted Alarm).** An alarm φ₁₂,V₂ is called an impacted alarm if

1) poi(φ₁₂,V₂) is unchanged and a similar alarm φ₁₁,V₁ was reported for the corresponding POI in V₁; and

2) at least one of the dependencies in cdDep⁺(poi(φ₁₂,V₂)) or cdDep⁺(poi(φ₁₁,V₁)) is modified.

Note that, in case (2), presence of a modified dependency is also checked in cdDep⁺(poi(φ₁₁,V₁)), because checking the presence of a modified dependency only in cdDep⁺(poi(φ₁₂,V₂)) does not capture modifications to dependencies through deletion of program statements in V₁.

For each impacted alarm φ₁₂,V₂, there exists a unique similar alarm φ₁₁,V₁ corresponding to it where l₁ and l₂ are the corresponding lines. We call these two alarms, φ₁₂,V₂ and φ₁₁,V₁, corresponding alarms.

We classify delta alarms into three classes, namely data-dependency impacted alarms (ddImpacted), value-dependency impacted alarms (vdImpacted), and control-dependency impacted alarms (cdImpacted). These classes are defined below.

**Definition V.6 (Data-Dependency Impacted Alarm).** Let φ₁₂,V₂ be an impacted alarm with φ₁₁,V₁ as its corresponding alarm. We call φ₁₂,V₂ a data-dependency impacted alarm if at least one of the dependencies in dDep⁺(poi(φ₁₂,V₂)) or dDep⁺(poi(φ₁₁,V₁)) is modified.

For example, A₁ in Section VI-B is a data-dependency impacted alarm (ddImpacted), because the data dependency of the index expression z in V₂, present at line 27, is modified.

Note that, for a ddImpacted alarm, the thin slices generated for it and its corresponding alarm are different.

**Definition V.7 (Value-Dependency Impacted Alarm).** Let φ₁₂,V₂ be an impacted alarm with φ₁₁,V₁ as its corresponding alarm. We call φ₁₂,V₂ a value-dependency impacted alarm if at least one of the following holds.

1) ∃ d ∈ vDep⁺(poi(φ₁₂,V₂)) such that d is a modified dependency and d /∈ dDep⁺(poi(φ₁₁,V₁)).

2) ∃ d' ∈ vDep⁺(poi(φ₁₁,V₁)) such that d’ is a modified dependency and d’ /∈ dDep⁺(poi(φ₁₁,V₁)).

In other words, we call an impacted alarm φ₁₂,V₂ a value-dependency impacted alarm (vdImpacted) if and only if at least one of the dependencies in vDep⁺(poi(φ₁₂,V₂)) or vDep⁺(poi(φ₁₁,V₁)) is modified.

We refer to an impacted alarm that does not belong to any of the above classes as a control-dependency impacted alarm (cdImpacted). An example of a cdImpacted alarm is A₂ in Section VI-B.

In other words, we call an impacted alarm φ₁₂,V₂ a control-dependency impacted alarm (cdImpacted) if and only if at least one of the dependencies in cdDep⁺(poi(φ₁₂,V₂)) or cdDep⁺(poi(φ₁₁,V₁)) is modified.

VI. RANKING OF DELTA ALARMS

In this section, we describe ranking of delta alarms obtained by prioritizing the six classes discussed in the previous section. We make the following observations for newly generated and impacted alarms.

a) Newly Generated Alarms: VSATs suppress repeated alarms that are not impacted by the changes between V₁ and V₂. If a newly generated alarm is suppressed, the alarm will remain suppressed on the subsequent versions, unless a code change in some next version impacts the alarm. If a newly generated alarm is an error, the error can persist in several next versions. Therefore, it is important to inspect each newly generated alarm when it is generated for the first time.

b) Impacted Alarms: The changes made between V₁ and V₂ generally correspond to fixing of bugs, addition of features, and refactoring. Failure to detect refactoring can result in generation of false impacted alarms, called spuriously impacted alarms (Section VI-C). Moreover, determining whether a code change (made to fix a bug or add a feature) impacts a POI is undecidable in general [37]. Hence the VSATs use conservative impact analysis that is based on data and control dependencies. As a consequence, expressions often get falsely
A. Prioritization of Newly Generated Alarms

A prioritization of newly generated alarms over impacted alarms.

B. Influencing Dependencies of Alarms

This section describes the notion of (non)-influencing dependencies of alarms, that we introduce and use to prioritize the three classes of impacted alarms. Recall that the dependencies in \( cdDep^+ (\alpha) \) correspond to the definitions and controlling conditions in backward slice generated for \( \alpha \), and vice versa. Thus, we refer to program statements on backward slice and dependencies in \( cdDep^+ (\alpha) \) interchangeably.

In general, a program statement corresponding to a dependency of an alarm is said to impact the alarm if the statement affects reachability of the alarm’s program point or determining whether the alarm is a false positive. That is, for an alarm \( \phi \), sound VSATs conservatively consider all dependencies in \( cdDep^+ (poi(\phi)) \) as impacting \( poi(\phi) \). As discussed in Section II-B2 not all dependencies in \( cdDep^+ (poi(\phi)) \) affect determining whether \( \phi \) is a false positive, i.e., some controlling conditions do not restrict values of variables in \( \phi \) but only control reachability of the program point where \( \phi \) is reported. Thus, to differentiate the impacting dependencies that only control reachability of alarms’ program points, from the other impacting dependencies that affect values of the variables in the alarms, we introduce the notion of non-influencing dependencies of alarms.

Definition VI.1 (Influencing dependency of an alarm). Let \( \phi \) be an alarm reported on a program \( P \), and \( d \) be a dependency of \( poi(\phi) \) (i.e., \( d \in cdDep^+ (poi(\phi)) \)). Depending on whether \( d \) is a definition, we distinguish the following two cases:

1) If \( d \) := \( x = e \) is a definition, let \( P' \) be a program obtained from \( P \) by replacing the RHS of the assignment expression (e) with a non-deterministic choice function. We say that the dependency \( d \) is an influencing dependency of \( \phi \) only if \( \phi \) is a false positive in \( P \) but an error in \( P' \). Otherwise, we say that \( d \) is a non-influencing dependency of \( \phi \).

2) If \( d \) := \( n \to n' \) is a conditional edge, let \( P' \) be a program obtained from \( P \) by replacing the condition of the branching node \( n \) with a non-deterministic choice function. We say that the dependency \( d \) is an influencing dependency of \( \phi \) only if \( \phi \) is a false positive in \( P \) but an error in \( P' \). Otherwise, we say that \( d \) is a non-influencing dependency of \( \phi \).

For example, assuming the three impacted alarms in \( V_2 \) are false positives, \( \tau_{24} \to \tau_{25} \in cdDep^+ (poi(\phi)) \) is an influencing dependency of \( \phi \) whereas the same dependency is a non-influencing dependency of \( \phi \).

Note that, if \( d \) is a non-influencing dependency of an alarm \( \phi \), all the dependencies in \( cdDep^+ (poi(\phi)) \) that impact \( \phi \) only through \( d \) are also non-influencing dependencies of \( \phi \). E.g., the definition on line 7 (resp. 21) is an influencing dependency of \( \phi \) (resp. \( \phi \)).

C. Spuriously Impacted Alarms

Recall that VSATs suppress non-impacting repeated alarms assuming that the user has inspected all alarms reported on the previous version and fixed identified errors (Section II-B). An impacted alarm needs to be manually inspected because its corresponding changes can result the same POI (for which a false positive was reported on \( V_1 \) into an error on \( V_2 \). We observe that such a conversion is possible only if the modified dependencies are influencing dependencies of the alarm. In the other case, i.e., when the modified dependencies are non-influencing dependencies, the impacted alarm is spurious. Identifying such spuriously impacted alarms helps to reduce the number of delta alarms reported to the user and hence the manual inspection effort.

Note that, the problem of computing spuriously impacted alarms involves computing (non)-influencing dependencies of alarms. This problem is undecidable in general, because the computation also requires determining whether the alarm is a false positive. Therefore, as discussed in the next section, we use an approximate method to identify (non)-influencing dependencies of alarms and use them to prioritize the three classes of impacted alarms.

D. Prioritization of Impacted Alarms

In this section, we rank the three classes of impacted alarms. We make the following observations from the prior work on program slicing [22, 31].

Observation 1: The backward, value, and thin slices generated for an expression are such that backward slice subsumes value slice, and value slice subsumes thin slice. Moreover, the size of value slice (resp. thin slice), in terms of the nodes on that slice, is on average about 50% (resp. 25%) of the size of backward slice [22, 31].

Observation 2: In their evaluation of value slice used in automated elimination of false positives, Kumar et al. [22] found the following.

(2.1) If thin slice is used instead of backward slice, 29% of alarms do not get eliminated as false positives.
(2.2) If value slice is used instead of backward slice, only 2%
of alarms do not get eliminated as false positives.

Based on observation (2.1) we can say that, for 29% (resp. 71%) of alarms, the dependencies in their \(vdDep^+\) are actually influencing (resp. non-influencing). In other words, for an impacted alarm \(\phi\), if a dependency in \(vdDep^+(poi(\phi))\) is modified, \(\phi\) is more likely to be a spuriously impacted alarm, as compared to an impacted alarm \(\phi'\) resulting due to modification to a dependency in \(dDep^+(poi(\phi'))\). Recall that an impacted alarm \(\phi\), with \(\phi'\) as its corresponding alarm, is a \(vdImpacted\) alarm iff at least one of the dependencies in \(vdDep^+(poi(\phi))\) or \(vdDep^+(poi(\phi'))\) is modified (Definition \(V.7\)). Thus, we can conclude that \(vdImpacted\) alarms are more likely to be spurious than \(ddImpacted\) alarms.

Based on observation (2.2) we can say that, only for 2% (resp. 98%) of alarms, the dependencies in their \(cdDep^+\) are actually influencing (resp. non-influencing). In other words, for an impacted alarm \(\phi\), if a dependency in \(cdDep^+(poi(\phi))\) is modified, \(\phi\) is more likely to be a spuriously impacted alarm, as compared to a \(vdImpacted\) alarm. Recall that an impacted alarm \(\phi\), with \(\phi'\) as its corresponding alarm, is a \(cdImpacted\) alarm iff at least one of the dependencies in \(cdDep^+(poi(\phi))\) or \(cdDep^+(poi(\phi'))\) is modified (Definition \(V.8\)). Therefore, we can conclude that \(cdImpacted\) alarms are more likely to be spurious than \(vdImpacted\) alarms.

Based on the observations above, we propose the following prioritization for the three classes of impacted alarms. \(ddImpacted > vdImpacted > cdImpacted\). Thus, the prioritization of the six classes of delta alarms is as following: \(Result-changed > POI-added = POI-changed > ddImpacted > vdImpacted > cdImpacted\).

Indeed, the proposed ranking scheme is useful only if the number of alarms in the highly prioritized classes should be smaller and the number of alarms in the lower prioritized classes should be larger. Therefore, in Section \(VII\) we empirically evaluate distribution of delta alarms in those six classes, and measure percentage of \(cdImpacted\) alarms.

E. Grouping of Impacted Alarms

For each class of impacted alarms, we group its alarms based on their modified dependencies, i.e., the corresponding code changes. With such grouping, alarms in a group can be inspected together, because they are generated due to the same reason(s). We expect the grouping will help to reduce the manual inspection effort. Evaluating the reduction in manual inspection effort due to the classification and grouping of delta alarms is out of scope of this paper.

VII. EMPIRICAL EVALUATION

In this section, we evaluate distribution of delta alarms into the proposed six classes and measure percentage of alarms that can be suppressed using the proposed ranking scheme.

A. Experimental Setup

1) Implementation: As a baseline, we implemented impact analysis-based VSAT \([13]\) (Section \(II-B\)) using analysis framework of a commercial static analysis tool, TCS ECA \([24]\). This tool is the same as the tool used in the pilot study (Section \(III\)). We preferred impacted analysis-based VSAT over VMV \([12]\) due to the following two reasons. First, inferring useful correctness conditions—sufficient or necessary conditions—as required by VMV is challenging \([39]\). Second, similarly to impact analysis-based VSAT, our classification technique also requires generating program dependence graphs to compute the three transitive closures of the dependencies (Section \(IV\)). Furthermore, the classification technique can be seen as an extension of the impact analysis-based VSAT.

We implemented the classification of delta alarms using the same analysis framework, where we enhanced the framework to create program dependence graphs corresponding to backward, thin, and value slices, and to access dependencies in those graphs. We used \textit{diff} to create a mapping of the code from two consecutive versions, required to implement the VSAT and the classification technique.

2) Selection of Applications and Alarms: Evaluation of the technique presented in this paper requires analysis of multiple versions of an application. We selected the four open source applications and their versions we used in the pilot study (Section \(III\)). Additionally, we randomly chose three more open source C applications from the list of 100 applications used by Cha et al. \([23]\), with the same constraints used to select the applications in the pilot study: application size should be greater than 10 KLOC and lesser than 20 KLOC (to limit the amount of analysis time), and at least two versions of the application should be available online. We restricted the number of applications to seven and the number of total versions selected to 59, because (1) compiling each version with appropriate macros for the analysis is a manual and time-consuming activity \(3\) and (2) analyzing the code, computing delta alarms and classifying them takes a considerable amount of time. Table \(II\) lists these applications, their total number of versions selected, and the first and last versions selected.

In total, we analyzed 59 versions using TCS ECA for AIOB verification property, and then computed delta alarms from tool-generated alarms. We selected AIOB as the only property in our evaluation, because (1) AIOB is one of the commonly used verification properties in evaluations of static analysis tools and techniques, and (2) the number of alarms generated for other properties on the selected applications were too less (e.g., division by zero) or too many (e.g., arithmetic overflow-underflow). For each application, Table \(II\) summarizes the number of tool-generated alarms, delta alarms, newly generated alarms, and impacted alarms. The summarized number of alarms is on the selected versions except the first version, i.e., the number of delta alarms generated on \(V_2\) (compared to \(V_1\)) + the number of alarms generated on \(V_3\) (compared to

\(^3\)Compiling and getting those 59 versions ready for static analysis took around 1.5 months’ effort of an experienced developer.
TABLE II: Experimental results showing the alarms in each of the classes of delta alarms.

<table>
<thead>
<tr>
<th>Application</th>
<th>Versions selected</th>
<th>Total alarms</th>
<th>Delta alarms</th>
<th>Newly generated alarms</th>
<th>Impacted alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total versions</td>
<td>First version</td>
<td>Last version</td>
<td></td>
<td></td>
</tr>
<tr>
<td>archimedes</td>
<td>15</td>
<td>0.08</td>
<td>2.00</td>
<td>6373</td>
<td>4183</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>328</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>343</td>
</tr>
<tr>
<td></td>
<td>auto-apt</td>
<td>3</td>
<td>0.322</td>
<td>178</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td>51.72%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51.72%</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>dict-gcide</td>
<td>2</td>
<td>0.481</td>
<td>192</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>4 (25.00%)</td>
</tr>
<tr>
<td></td>
<td>gzip</td>
<td>12</td>
<td>0.73</td>
<td>803</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>400</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>160 (40.00%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>160 (40.00%)</td>
<td>372</td>
</tr>
<tr>
<td></td>
<td>gzip</td>
<td>9</td>
<td>1.39</td>
<td>1514</td>
<td>1446</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1446</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>223</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>224</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>224</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>59</td>
<td>1118</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1118</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>45</td>
<td>03.11%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>03.11%</td>
<td>1222</td>
</tr>
<tr>
<td></td>
<td>rhash</td>
<td>15</td>
<td>1.24</td>
<td>245</td>
<td>207</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>207</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>113</td>
<td>54.59%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>54.59%</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>sm-utls</td>
<td>3</td>
<td>0.96</td>
<td>484</td>
<td>484</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>484</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>255</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>227</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>227</td>
<td>46.09%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>46.09%</td>
<td>229</td>
</tr>
<tr>
<td>Grand Total</td>
<td>9789</td>
<td>6910</td>
<td></td>
<td>0</td>
<td>856</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>856</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>872</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>872</td>
<td>299</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>299</td>
<td>2053</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2053</td>
<td>3686 (53.34%)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3686 (53.34%)</td>
<td>6038</td>
</tr>
</tbody>
</table>

Taking a closer look at the results we observed that, the changes made between two consecutive versions varied greatly across the applications as well as their versions. During our analysis we also found that, quite often a single change to a control dependency was a reason to report many repeated alarms as impacted. For example, between 0.1.1 and 0.1.2 versions of archimedes application, the code on lines 125 to 130 in readinputfile.h is changed as shown below: the line shown with \++\ is newly added in version 0.1.2.

```c
else if(strcmp(s,"ALSB")==0) type=ALSB;
++
else if(strcmp(s,"INSB")==0) type=INSB;
else {
    printf("\%s : unknown specified material!\n", programe);
    exit(0);
}
```

Due to the above code change, 389 repeated alarms get reported as impacted, and 365 and 24 of them respectively are cdImpacted and vdImpacted. Since all those cdImpacted alarms are not dependent on the modified type variable, the newly added dependency (controlling condition) is a non-influencing dependency of those alarms. Thus, the alarms can be safely suppressed. We observed several such cases on archimedes and sm-utls applications. This indicated usefulness of the proposed technique to identify and suppress a large number of spuriously impacted alarms and reduce the required manual inspection effort.

To validate (non)-influencing dependencies computed for alarms in vdImpacted and cdImpacted classes, we randomly selected 50 alarms from each class and manually inspected all their dependencies. We found that, for all those selected alarms, the dependencies that were identified as non-influencing were indeed non-influencing. That is, all those alarms were reported as impacted due to the conservative impact analysis. Suppressing such impacted alarms will not result in a false negative.

Indeed, evaluating the proposed ranking scheme requires to compute percentage of false positives/true positives for alarms in each of the classes. During our closer look of the analysis results, we found that the selected versions, being well tested applications, are not suitable to compute the percentages. We made attempts to obtain a set of delta alarms that are

\( V_2 \) and so on. The alarms generated on the first version are not a part of this table, because analysis of this version is not version-aware.

B. Results Discussion

1) Distribution of Delta Alarms: We applied the classification technique to delta alarms generated on the selected versions. Table II presents the number of alarms in each of the six classes. Inspecting the table, we make the following observations. 1) Around 70% of tool-generated alarms get reported as delta alarms; the remaining alarms are suppressed by the impact analysis-based VSAT. 2) The impacted alarms dominate the newly generated alarms: around 87% delta alarms are impacted while the remaining 13% are newly generated. 3) Majority (98%) of the newly generated alarms belong to POI-added class, whereas five out of the seven applications had no POI-changed alarms at all. 4) Among impacted alarms, only 5% are ddImpacted, while 34% and 61% respectively are vdImpacted and cdImpacted. This indicates that cdImpacted alarms dominate the other alarms in their number.

2) Ranking of Delta Alarms: Recall that, in the proposed ranking scheme of delta alarms, result-changed alarms are assigned the highest priority (Section VI-A). The evaluation results (Table II) indicate that no newly generated alarm is a result-changed alarm. A possible reason to this could be that, these applications being well tested, no such (AIOB) error existed in these applications.

Since cdImpacted alarms are most likely to be spuriously impacted alarms (Section VI-0b), they can be suppressed if required. Thus, overall, the proposed ranking allows to identify around 53% of delta alarms as more likely to be false positives than the others. Note that percentage of the suppressible alarms vary greatly as per the applications, from 3% (gzip) to 73% (archimedes). The median reduction that can be achieved by suppressing cdImpacted alarms is 48.9%. Overall, the percentages of delta alarms belonging to each of those six classes are: 0% (result-changed), 12.4% (POI-added), 0.2% (poi-changed), 4.3% (ddImpacted), 29.7% (ddImpacted), and 53.3% (cdImpacted). This indicates the distribution of delta alarms into those six classes is as it is expected for the ranking scheme to be useful (Section VI-D).
labelled as true positives and false positives, and could find none. Since creating such a labelled data is effort-intensive task and can introduce bias, as a future work we plan to conduct a controlled study in this direction. Note that, the identification of cdImpacted alarms as suppressible is based on the confirmed findings from the evaluations of the three types of code slices.

3) Grouping of Alarms: We found that often the number of groups of impacted alarms in the three classes are very few as compared to the total number of grouped alarms. For example, there are 415 cdImpacted alarms generated on 2.0.0 version of archimedes, and they are grouped into 20 groups. As another example, the 365 cdImpacted (and 24 vdlImpacted) identified due to the code change example discussed above in Section VII-B2 are grouped together. As a consequence, those 365 cdImpacted can be manually inspected or suppressed together. We expect that the proposed classification, grouping, and ranking will help to reduce the inspection effort.

4) Threats to Validity: The effectiveness of the proposed ranking scheme needs to be evaluated based on the percentages of true positives in each of those six classes. However, due to the unavailability of the dataset, we argued the effectiveness of the ranking scheme based on the observations made from the prior work on program slicing. Performing a controlled experiment can change the findings. Threats to internal validity concern the extent to which the observations are correctly derived from the experimental data. In our evaluation, threats to internal validity concern selection of alarms, applications, VSAT, and implementation to compute the three types program slices. We used 59 versions of seven applications, however restricted number of verification properties and VSATs is a main concern. Threats to external validity concern the extent to which the evaluation results generalize beyond the sample used (the alarms and techniques selected in the evaluation) to the entire population. Since our evaluation is using restricted alarms (single property and fewer applications), the findings made may not generalize to entire population and is a threat to the validity.

VIII. RELATED WORK

In this paper, we proposed a technique for classification and ranking of delta alarms. Thus, we compare it with alarms postprocessing techniques which employ similar approaches, namely pruning/classification, and ranking [16].

a) Pruning/Classification of Alarms: The techniques in this category classify alarms mainly into two classes, actionable and non-actionable [15], [16]. The non-actionable alarms being more likely to be false positives, they are not reported to the user. The techniques vary based on the methods they employ to achieve the classification, and a majority of the techniques are based on machine learning [40], [41]. The version-aware static analysis techniques (VSATs) [12], [14], [19], [20] also belong to this category as they suppress a subset of the alarms generated, calling them as non-impacting or not important. As discussed in Section II-B, unlike VSATs, our technique uses the code changes, due to which the delta alarms are generated, to postprocess those alarms further.

Although our ranking and pruning technique is designed to postprocess delta alarms independently of the techniques generating them, it can also be applied on its own: the input to the technique can be the tool-generated alarms instead of delta alarms. To the best of our knowledge the other classification techniques do not use the code changes between the versions and relate them with the generated alarms.

b) Ranking of Alarms: The existing techniques to rank alarms employ different approaches such as statistical analysis, history of the bugs and alarms fixing, and even feedback from the user [11], [16]. Among them, the techniques that are based on history of fixing of alarms [42], [43] and bugs [44] prioritize alarms by analyzing software change history. Thus, our technique is similar to them. However, the underlying method to prioritize alarms is different: these techniques analyze the change history to mine commonly/quickly fixed alarms and bugs, while our technique is based on the causal relationship and thus is orthogonal to them. Heo et al. [45] have proposed a technique to rank alarms generated on evolving code. As the alarms in the proposed six classes can be still large in number, they can be further ranked using the other ranking techniques.

Furthermore, our introduced notion of non-influencing dependencies of alarms is similar to non-impacting controlling conditions proposed by Muske et al. [46]. However, our notion is applicable to definitions (assignment statements) too.

As discussed above, our proposed classification and ranking technique is orthogonal to the existing classification and ranking techniques [11], [16]. Thus, they can be combined with the existing techniques to obtain more benefits as compared to the benefits obtained by applying them individually.

IX. CONCLUSION AND FUTURE WORK

In this paper, based on our observation that the existing version-aware static analysis techniques do not use code changes to further postprocess delta alarms, we proposed a technique for classification and ranking of delta alarms. The technique classifies delta alarms into six classes based on the type of changes due to which they are identified as delta alarms. It then ranks the alarms by assigning different priorities to postprocess delta alarms independently of the techniques generating them, it can also be applied on its own: the input to the technique can be the tool-generated alarms instead of delta alarms. To the best of our knowledge the other classification techniques do not use the code changes between the versions and relate them with the generated alarms.

We plan to evaluate the reduction in effort achieved due to the proposed classification, ranking, and grouping of delta alarms. Towards this we will perform a controlled study by involving multiple participants who are (experienced) users of static analysis tools. Moreover, we plan to conduct evaluation by using alarms generated for other verification properties and a few more additional applications.