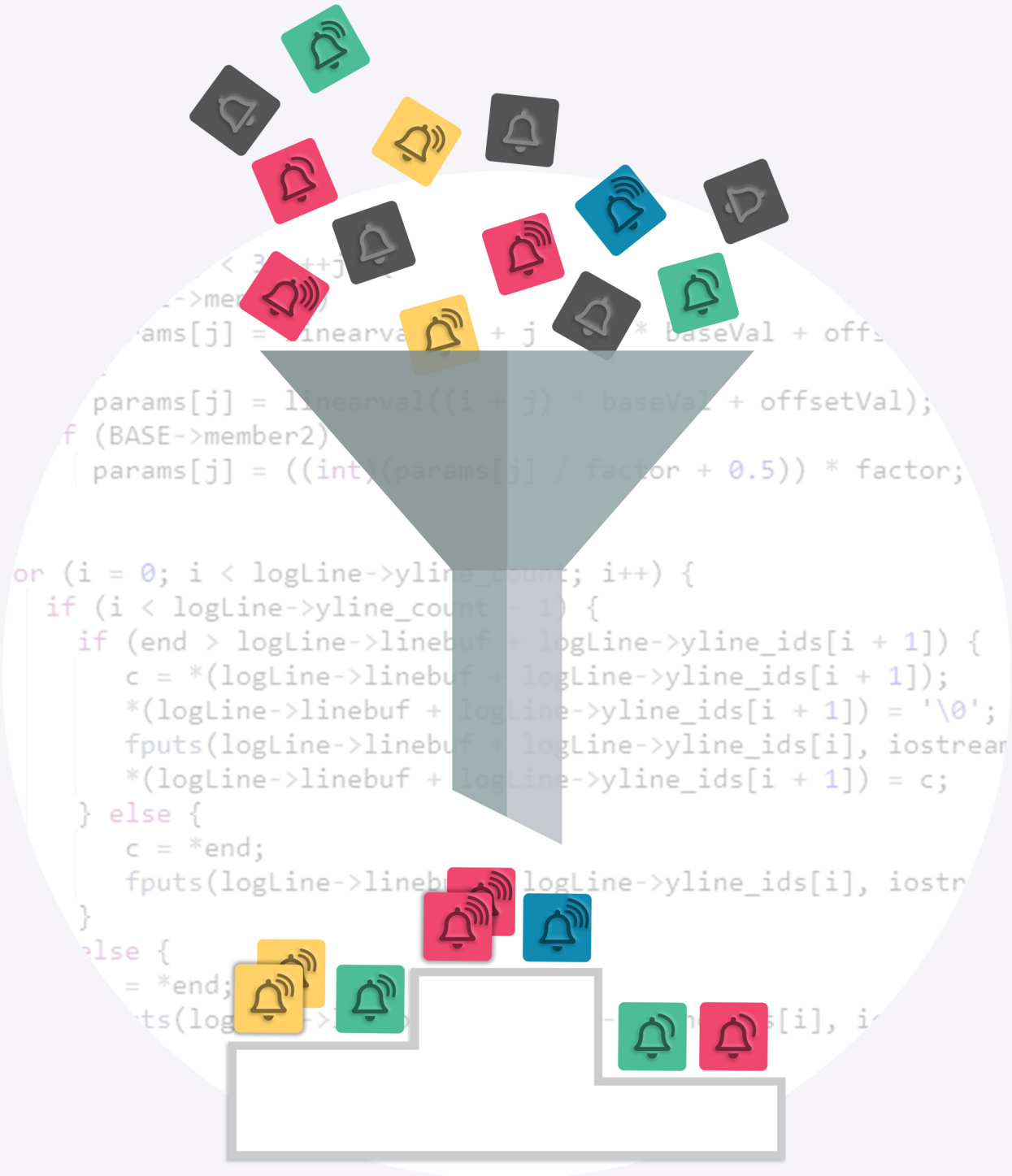


Postprocessing of Static Analysis Alarms

Tukaram Bhagwat Muske

Postprocessing of Static Analysis Alarms

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties, in het openbaar te verdedigen
op dinsdag 7 juli 2020 om 13:30 uur

door

Tukaram Bhagwat Muske

geboren te Hipparsoga, Maharashtra, India

Dit proefschrift is goedgekeurd door de promotoren en de amenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. J.J. Lukkien
1e promotor:	dr. A. Serebrenik
2e promotor:	prof.dr. M.G.J. van den Brand
leden:	prof.dr.ir. J.P.M. Voeten
	prof.dr. S. Schupp (Hamburg University of Technology)
	prof.dr. Y. Smaragdakis (University of Athens)
adviseurs:	dr.ir. T.A.C. Willemse
	MSc. R. Venkatesh (Tata Consultancy Services, India)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Postprocessing of Static Analysis Alarms

Tukaram Bhagwat Muske

First promotor: Dr. A. Serebrenik (Eindhoven University of Technology)
Second promotor: Prof. Dr. M.G.J. van den Brand (Eindhoven University of Technology)

Additional members of the reading committee:

Prof. Dr. S. Schupp	(Hamburg University of Technology)
Prof. Dr. Y. Smaragdakis	(University of Athens)
Prof. Dr. ir. J.P.M. Voeten	(Eindhoven University of Technology)
Dr. ir. T.A.C. Willemse	(Eindhoven University of Technology)
MSc. R. Venkatesh	(TRDDC, Tata Consultancy Services)

The work in this thesis has been fully funded by Tata Consultancy Services Limited, India.

TATA CONSULTANCY SERVICES

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-5037-1

An electronic version of this dissertation is available at
`https://research.tue.nl/` and
`https://tmuske.github.io/phdThesis/`

Cover design: Anushri Jana & Ankur Jana

Printed by: ProefschriftMaken ll `www.proefschriftmaken.nl`

Dedicated to…

Vitthal Wangwad

my grandfather,

without whom my education would have not been possible.

Acknowledgements

And the time has come to express my deep gratitude and sincere thanks to all those who motivated me to undertake this journey for a PhD, who helped me along the journey, and who contributed in an invaluable way to the completion of the thesis. The list is very long!

Taking the path for a PhD requires a lot of motivation and clarity. After having settled as an applied researcher in TRDDC, TCS, for about six years, I was still lacking these. Uday Sir (my mentor, and a professor at IIT Bombay), and Ulka and Venky (my supervisors at TCS), you were those who kept motivating me to do a PhD. The process, to motivate me, started sometime in 2011, and went on for almost five years. Without your constant push, I wouldn't have taken this path. And not just for motivating, I am also grateful to you for being there always for me whenever I required any guidance and support. You have always given me more than what I have asked for, and the same support continues even now! You were instrumental in setting up the stage and ensuring I reach here. Venky and Ulka, I have always enjoyed being supervised by you, and am thankful to you for leading me in a fruitful direction and allowing me the freedom to explore them. Ananth (CTO at TCS) and RD (TRDDC Head), I can't miss mentioning you! Your opinion and feedback was crucial in making my decision to go for the PhD. Towards the end of 2015, the decision was made. So, the time had arrived, but not the place!

The place for my PhD arrived (rather, I arrived at the place) when I visited TU/e for ETAPS 2016. It was there that I met you both Mark and Alexander (from TU/e). The way you explained the process and available options, and offered me the unique opportunity, the decision for the place was immediate! And the journey started. Since then, you both have been wonderful, friendly, and encouraging PhD supervisors (which, I later discovered, was a unanimous feeling among the PhD students at TU/e). You have helped me become a better person, both academically and personally. Alexander, if I express my gratitude to you in words, the expression will not do justice to the support, mentoring, and guidance that I have received from you, because no words can express it. Mark, thank you so much for finding time from your busy schedules and providing valuable feedback from time to time. I have enjoyed discussions with you, and have always benefited from your insightful suggestions and vast experience. Also, I cannot forget the warm welcomes and send-offs you both gave whenever I visited TU/e. I could see the effort that you both put in, to make me feel at home and ensure that I get enough exposure and learning experience through interactions with the people from within and outside the group. Thank you both, again, for making my PhD journey a very smooth and joyful ride.

Before I continue to thank everyone who helped me during the journey, I would like to pause and thank, sincerely, all the members of my thesis review committee. Prof. Lukkien, Prof. Voeten, Prof. Schupp, Prof. Smaragdakis, including Tim, Venky, Mark, and Alexander, your detailed and insightful comments have improved the thesis remarkably. Moreover, your comments helped me gain new insights into the work and look at it from different perspectives. I truly appreciate the time and effort you put in to review the thesis.

Now its time to thank those who were there for me whenever I required anything officially or personally. Hari and Samina (from our HR team at TCS), I appreciate the way you helped me by taking care of so many formalities, including the ones necessary to arrange my visits to TU/e and Alexander's visits to TRDDC.

Shrawan (a senior from our group at TCS, and a close mentor), please accept my deepest thanks for guiding me in the implementation and evaluation of the techniques included in the thesis. Metta (a senior in my group at TCS), I don't know how to express my gratitude to you, man! You have always been there as a friend, colleague, and personal mentor. Madhukar (my desk-mate at TCS and, more importantly, a friend), I am grateful to you for being there beside me all the time and helping me wherever I required your expertise in several things. Komal and Divyesh (my friends at TCS, and also cadets, like me, in PhD program), thanks to you as well! I loved and enjoyed the PhD related discussions, and appreciate your help whenever I asked for.

Advaita, Bharti, Prasad, Priyanka, Anushri, Amey, Supriya, and everyone in the *Foundations of Computing* group at TRDDC, you all have been such amazing friends and have helped me in many ways. Avriti, thanks a lot for helping me to set up the tool chain required for evaluating the AFPE techniques. Shivayu, Rohith, Vishnu Priya, and Abhinav, thank you for making my PhD life a lot easier. Without you, the work in the thesis would not have been completed. Peeyush, Anushri Laddha, Nathan van Beusekom, and Prajkta, thanks for helping me with the implementations and evaluations of the techniques during your internships at TRDDC. Ankur and Anushri, I sincererly appreciate your help in designing the cover page of the book.

Paddy (a close friend from Oracle Labs in Brisbane), I can't miss this opportunity to thank you and appreciate your timely reviews of papers even at my last-hour requests. Bonita (a collaborator from the University of Nebraska - Lincoln), thanks a lot for taking us down the lane of exciting work related to the eye tracking study and providing the collaboration opportunity.

Now, I would like to thank all the wonderful people in SET and FSA groups at TU/e. Anton, Tim, Yanja, Eleni, Tom, Jan Friso, Jurgen, Bas, Loek, Ion, Michel, Erik, Ruurd, Yaping Luo, thanks a lot for the technical discussions and the nice conversations we had, whenever I was there. Mahmoud, you were simply amazing! I am so lucky to have found you there! You made me feel at home during my visits to TU/e. You have provided me a bundle of great memories, especially bicycling around the park. Priyanka and Sangeeth, you have been my best buddies. You took care of so many things, and always made my stay in Eindhoven enjoyable and memorable. Josh, Weslley, Sander, Felipe, Miguel, Dan, Fei, Rick, Nan, Mauricio, Arash, Onder, Maurice, Kousar, Jouke, Rodin, Alexander (Fedorov), Muhammad, Nathan, Omar, Olav, I am very thankful to you guys too! I immensely enjoyed the time I spent with you people. Thomas, I cannot describe how overjoyed I was when you arranged the send-off party and cooked the pancakes. Also, you have been my first stop whenever I required any details about the formalities at TU/e. Thanks for all the wonderful memories that you have given me. Margje and Agnes, thanks a lot for taking care of everything during my visits. You both have been two of the most wonderful and lovely people I have met. Ege (IE&IS group), thanks for getting me involved in those outing and dinner plans.

I want to thank all my friends whose names I could not mention explicitly. You have been a source of constant support, and I know that you will always be! I also wish to thank my family from the core of my heart, especially my brother, Sandipan, for these words of you – *"I want to see you as Dr. Tukaram Muske"* – which stopped me from abandoning this journey when my confidence shook a little, once in the middle. Venky, you were also instrumental then. Thanks!

Tukaram Bhagwat Muske
May 30, 2020

Summary

Postprocessing of Static Analysis Alarms

Static analysis tools help to automatically detect common programming errors like *division by zero* and *array index out of bounds*, as well as help to certify absence of such errors in safety-critical systems. When these tools cannot determine whether a point of interest is an error, they report an *alarm*, i.e., a warning message notifying the tool user about a potential error at that point. Due to several reasons, such as the abstractions the tools use and the trade-off they make between precision and scalability of the analysis, they generate a large number of alarms. The user is required to manually inspect those alarms and partition them into errors and false positives. The large number of false positives and the effort required to manually inspect them have been reported as two primary concerns associated with limited adoption of static analysis tools in practice.

While these concerns can be addressed by improving precision of the analysis, the improvement achieved in this way is limited due to the inherent limitations of static analysis. An alternative, explored since last decade and half, is *postprocessing of alarms*: processing alarms after they have been generated. The postprocessing goals include reducing the number of alarms and effort required for their manual inspection. Considering the benefits offered by postprocessing of alarms, a plentitude of techniques have been proposed. However, percentage of alarms remaining after applying those techniques is still large, ranging between 40% to 80%. Moreover, even after simplification of manual inspection of alarms by the techniques, the alarms require considerable effort to inspect them manually (ranging between three to eight minutes per alarm). Therefore, *in this thesis we focus on improving postprocessing of alarms.*

As a starting point for our work we conduct a *comprehensive review* of techniques that have been proposed for postprocessing alarms. Indeed, while multiple postprocessing techniques have been proposed, such a review was missing making it harder to understand relative strengths and weakness of different techniques or to identify promising directions for future research. We identify 130 primary studies that propose techniques to postprocess alarms, and categorize the approaches implemented by them into six main categories. We then select and study the techniques and approaches that are useful when static analysis tools are used for proving absence of errors. Based on this study, we identify limitations of the existing techniques. We aim to improve alarms postprocessing by overcoming those limitations.

Clustering of alarms, one of the identified six categories of the approaches, is commonly used to reduce number of alarms. We find that state-of-the-art clustering techniques fail to group similar alarms appearing in commonly occurring scenarios. Therefore, we choose to improve clustering of alarms, and propose *repositioning of alarms* as means to overcome the limitation

of the clustering techniques. Repositioning reduces the number of alarms by moving groups of similar alarms along the control flow to a program point where they can be replaced by a single alarm. Our empirical evaluation indicates that the alarms repositioning technique reduces the number of alarms by up to 20% over state-of-the-art alarms clustering techniques with median reduction of 7.25%.

While we expected repositioning to reduce the number of alarms significantly, the median reduction is limited. This is why we take a closer look at reasons for the limited reduction in alarms by our technique. We find that a high percentage of similar alarms are not grouped together due to the conservative assumption related to conditional statements. To improve the repositioning technique and thus further reduce the number of alarms, we introduce a notion of *non-impacting control dependencies* (NCDs) and propose a new variant of repositioning based on NCDs. We evaluate the NCDs-based repositioning on 16 open-source and 16 closed-source systems. The evaluation indicates that, compared to the previous repositioning technique, the NCDs-based repositioning reduces the number of alarms on an average by up to 36.09%, with a median reduction of 10.48%.

Then we turned our attention to challenges incurred in static analysis of large systems. A popular way of scaling up the analysis consists in splitting an application code into multiple partitions. Each code partition is then analyzed separately, conservatively assuming all values being possible for variables shared by multiple partitions. This approach, however, increases the number of alarms because multiple alarms can get generated for the same *point of interest* when it belongs to multiple partitions (common-POI alarms). We find that postprocessing and manual inspection of common-POI alarms separately in each of their associated partitions result in redundancy. To reduce the redundancy problem in manual inspection, we group common-POI alarms together and propose a method to manually inspect them based on an automatically inferred condition for each group. We then target reducing redundancy in *automated false positives elimination* (AFPE) applied to common-POI alarms and reduce the time taken by AFPE. To this end, we reuse AFPE results across partitions. Our empirical evaluation indicates that (1) the proposed method to group and inspect common-POI alarms reduces manual inspection effort by 60%; and (2) the reuse of AFPE results across partitions reduces AFPE time by up to 56%, with median reduction of 12.15%.

Last, we aim to improve postprocessing of *delta alarms* that are generated by *version-aware static analysis tools* on evolving software. The improvement is based on our finding that postprocessing and reporting of delta alarms can be further improved by taking into account the code changes between the versions. However, none of the existing VSATs postprocesses delta alarms based on the code changes. In our proposed postprocessing, we classify delta alarms into six classes and rank them by assigning different priorities to these classes. The ranking of alarms can help to suppress the alarms that are ranked lower when resources to inspect all the alarms are limited. Next, we postprocess the ranked delta alarms for AFPE. We use the code changes to determine situations where AFPE results from the previous version can be reused. The reuse of AFPE results helps to improve efficiency of AFPE applied to delta alarms. Our empirical evaluation indicates that the proposed classification and ranking of delta alarms help to suppress 61% of delta alarms. The reuse of AFPE results across the versions reduces the AFPE time by 64.5%.

The presented techniques in this thesis are tool-agnostic and they can be applied in conjunction with other existing alarms postprocessing techniques to complement each other. We believe that the combinations will provide more benefits as compared to the benefits obtained by applying them individually. Investigating different combinations to obtain optimal results should be subject of further studies.

Table of Contents

Chapter 1

Introduction

In this chapter, we first introduce static analysis and alarms generated by static analysis tools (Section 1.1). We then describe the problem of large number of alarms and their postprocessing as an approach to address the problem (Section 1.2). Next, we briefly discuss limitations of techniques proposed for postprocessing of alarms, that we identify and address in this thesis. In order to overcome these limitations, we formulate a series of research questions (Section 1.3). Lastly, we provide outline of this thesis (Section 1.4).

1.1 Background

Software systems are playing a crucial role in our life. Ensuring that these systems are free of defects is of utmost importance for uninterrupted use of the systems or even for human lives in case the systems are safety-critical[1]. Software testing is commonly used to assess whether a software system or its components conform to their requirements. Despite its common use, software testing can be used to show the presence of bugs, but never to show their absence [53].

This well-recognized limitation of testing led to the emergence of alternative techniques such as static analysis, model checking, and symbolic execution, for ensuring system correctness. These techniques, commonly called *automated program analysis techniques*[2], are useful in the cases where a software system requires proving it to be free of defects of certain types [24, 52, 57, 112]. For example, these techniques can help to ensure that a safety-critical system does not crash due to common programming errors like *division by zero* or *dereference of a null pointer*. As another example, the techniques can help to prove absence of common vulnerabilities

[1] Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment [109].

[2] The terms used for the analysis techniques in scientific literature vary. Some studies (e.g. [229]) consider model checking and symbolic execution as a part of static analysis, while others (e.g. [57, 60, 153]) consider them to be different. Throughout this thesis, we consider static analysis to be different from model checking and symbolic execution. By static analysis, we mean analysis of programs to detect programming errors using techniques such as data flow analysis [100], abstract interpretation [38], and pattern matching [14].

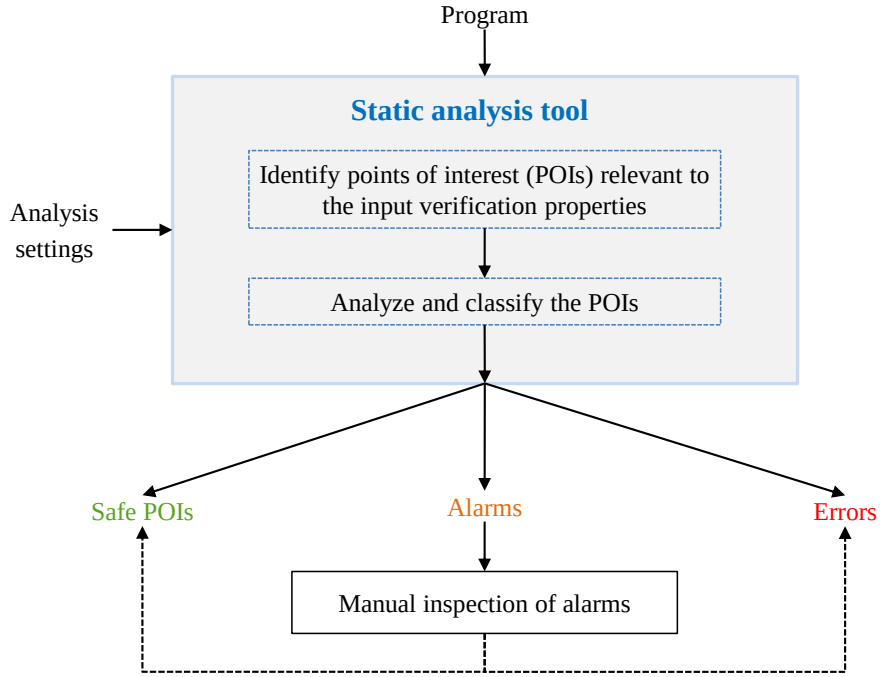


Figure 1.1: Working and usage of a typical static analysis tool.

in web applications, by checking whether values accepted from untrusted environment can flow into security-sensitive operations [170, 203, 204]. We use *code proving* to refer to such instances in which ensuring absence of defects of certain types is required. Apart from code proving, the automated program analysis techniques also help to detect defects early in the software development life-cycle [12, 14, 59]. For example, FindBugs [12] is a very popular static analysis tool used to find bugs in Java programs. We use *bug finding* to refer to instances in which the analysis purpose is to find bugs rather than to prove their absence.

Different automated program analysis techniques are known to have different strengths and limitations [57, 229]. For example, compared to model checking and symbolic execution, static analysis can scale to very large systems but can be less precise [153, 174, 205].

1.1.1 Static Analysis Tools

A wide range of *automated static analysis tools* (ASATs)[3] have been developed by the research community and industry. The tools developed vary in numerous aspects such as techniques they use to analyze the code, types of programming errors they detect, programming languages they support, and analysis purpose they have (code proving or bug finding). Wikipedia[4] lists more than 100 static analysis tools, such as Astrée [40, 51], Polyspace Code Prover [200], FindBugs, and ANDROMEDA [203].

[3]Throughout the thesis, we use static analysis tool(s) and the acronym ASAT(s) interchangeably. We drop the word *automated* when we explicitly refer to static analysis tool(s). We considered *automated* in the acronym to avoid its conflict with SAT which has other well accepted connotation.

[4]`https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`

Figure 1.1 provides a generalized view of working of an ASAT and workflow associated with its usage. A typical ASAT takes as input program code to be analyzed and *analysis settings* that configure or control the way code is analyzed. We categorize working of ASATs into two parts, and describe it below in Sections 1.1.1.1 and 1.1.1.2.

1.1.1.1 Identification of Points of Interest (POIs)

We refer to the type of errors that are detected by ASATs as *verification properties*. Usually the verification properties are provided as a part of input analysis settings. During analysis of the code, ASATs first identify expressions in the program where defects corresponding to the provided verification properties can manifest. We call those expressions *points of interest*[5] (POIs) relevant to the verification properties. The POIs identified by an ASAT vary depending on whether the ASAT supports bug finding (a bug finding tool) or code proving (a code proving tool). A bug finding tool generally identifies a subset of POIs that are relevant to a given verification property. For example, FindBugs identifies POIs for *dereference of a null pointer* verification property only if they appear in certain code patterns, such as dereference of a reference variable is without a null check and there exists another dereference of the same variable but with a null check. A code proving tool, such as Polyspace Code Prover and Astrée, selects all the POIs that are relevant to a given verification property. For example, these tools identify *every dereference of a pointer* as a POI relevant to *dereference of a null pointer* verification property.

1.1.1.2 Analysis and Classification of POIs

After identification of POIs corresponding to input verification properties, ASATs analyze the code using techniques such as *data flow analysis* [100, 163], *abstract interpretation* [38], and *difference bound matrix* [146]. We call these techniques *value analysis techniques*, because they compute values of variables at some or all program points in the program. Using the analysis results, ASATs primarily intend to classify each POI either as *safe* or *erroneous*, depending on whether the POI violates the property. As determining whether a POI is safe is undecidable in general [54, 118, 178, 182], in several instances ASATs cannot classify a POI either as safe or erroneous. In such a case, they conservatively report an *alarm* for that POI. An alarm reported by ASATs is a warning message to the user denoting a *potential error* at its corresponding POI.

Usually this step of *analysis and classification of POIs* in ASATs is absent if they are based on pattern-matching [14]: pattern-matching is used for both identification and classification of POIs. Computation of values of variables by the value analysis techniques can be *shallow or deep* depending on the requirement [59, 80]. An analysis that is typically *flow- and context-insensitive* is called *shallow analysis*, whereas an analysis that is *flow- and context-sensitive* is called *deep analysis* [59]. The ASATs that are based on shallow analysis or pattern-matching are generally called *shallow/light-weight analysis tools*, whereas the other type of ASATs are called *deep analysis tools* [59, 207]. In general, a bug finding (resp. code proving) tool is shallow (resp. deep) analysis tool. □

Manual Inspection of Alarms As alarms generated by ASATs are warning messages about potential errors, the user is expected to inspect each of them and decide whether the alarm's POI is erroneous. An alarm whose corresponding POI does not indicate an error is commonly called a *false positive*. We find that, however, the terms used in literature to refer to alarms that indicate

[5]Note that the points of interest (POIs) are different from the *program points*. A program point can have multiple POIs related to a verification property.

errors vary, e.g., true positives [179, 210], actionable alarms [77, 185], and true errors [182]. Throughout this thesis, we use the term *error* to refer to an alarm that is a true positive, i.e., the one whose POI violates the property.

1.1.2 Static Analysis Alarms

ASATs are known to generate a large number of false alarms [37, 92, 120]. Below we describe a few prominent reasons for generation of a large number of false alarms by ASATs.

1.1.2.1 Use of Abstractions

The value analysis techniques used by ASATs, mainly deep analysis tools, generally compute values of all variables in the program and at all program points. Such a computation is memory and time intensive in presence of large-size arrays, data structures like linked-lists and map-tables, and recursive chains. Therefore, in these cases the tools use abstractions, i.e., they compute more possible values for variables than the variables can take [56, 182]. When more values are computed for a variable in a POI than the variable actually can take, it results in imprecision of the tools: a false positive is most likely to get generated for the POI due to the more values computed for the variable.

1.1.2.2 Lack of Run-time Context

Since ASATs analyze a program without executing it, in many cases the exact run-time values are not available due to, e.g., user input or values from external data sources. To address this issue, ASATs adopt conservative approach, i.e., assume that all values are possible when an input is accepted from user or values are read from external data sources. The conservatively assumed values increase the number of alarms generated by ASATs.

1.1.2.3 Trade-off between Precision and Scalability

In general, analysis performed by ASATs, especially by deep analysis tools, is memory and time-intensive. Hence, the tools generally do not scale on very large systems. In such cases, the tools might compromise on precision of the analysis to improve scalability. As a result of this trade-off, more alarms get generated. Generally, this trade-off is resolved by developers of ASATs or by users of ASATs. For example, Polyspace Code Prover allows the user to select from different levels of precision which correspond to different degrees of scalability.

As another example, to scale up static analysis on very large systems (ranging in millions lines of code), a deep analysis ASAT might split an application code into multiple parts (called *code partitioning*) [89, 99]. Each code part, called *partition*, is then analyzed separately under the conservative assumption that all values are possible for variables shared by multiple partitions. This conservative assumption affects precision of the analysis. Moreover, multiple alarms also can get generated for the same POI when the POI belongs to multiple partitions. Thus, although a code partitioning approach helps ASATs to analyze very large systems, it results in generating more alarms on the partitioned-code than the number of alarms that would be generated on the non-partitioned code.

1.2 Setting the Context

1.2.1 The Problem

Due to the reasons discussed above (Section 1.1.2), generation of alarms by ASATs is certain, their number can be large, and they require manual inspection to identify errors among them. The number of alarms generated and percentage of false positives among them vary depending on precision of the tool. Moreover, manual effort required to inspect alarms varies depending on several factors such as expertise of the user, verification properties, and tool support used [54, 150]. To describe the problems associated with alarms, we provide the following findings from a few studies.

- In general there are 40 alarms for every thousand lines of code [20, 82].
- Studies by Kamperman [98], and Beller et al. [20] report that there can be as many as 50 false positives for every accurately reported alarm (error). Moreover, based on studies relevant to static analysis alarms, Heckman and Williams [84] report that 35% to 91% of reported alarms are false positives.
- Manual inspection of alarms is a tedious and time-consuming process [54, 150, 182, 195]. Beller et al. [20] based on their literature survey report that, on average, inspection of an alarm takes three to eight minutes.
- In addition to the above problems, manual inspection of alarms is also found to be *error-prone* [54].

1.2.2 Postprocessing of Alarms

Popularity of ASATs in the industry is found to be varying, mainly based on type of the tools (code proving or bug finding) and the nature of applications to be analyzed. On one hand, several studies report that, despite a large number of alarms generated and manual inspection required for them, ASATs are often used to prove absence of certain types of defects. However the use is found to be mainly limited to safety-critical and security-critical systems. That is, ASATs are used when conventional techniques of software testing are very costly or inadequate to detect those critical defects, or the usage of such tools is mandated by certification agencies [24, 52, 112]. The problem of large number of alarms is still evident during the usage of tools and addressing it is a challenge. On the other hand, several studies [20, 37, 92, 120] suggest that the large number of alarms generated and effort required to manually inspect them are two major concerns during adoption of ASATs in practice. Johnson et al. [92], and Christakis and Christian [37] report that ASATs are underused in practice. Combining the insights from the two groups of studies, we conclude that ASATs are not commonly used by developers except for safety- and security-critical systems.

Therefore, to help users during the use of code proving ASATs and to increase adoption of bug finding ASATs, the problem of large number of alarms needs to be addressed. The obvious approach to address this problem is to improve precision of ASATs, and this line of work is being pursued by researchers [17, 75, 135]. However as discussed earlier (Section 1.1.2), due to several commonly occurring reasons, reporting of false alarms by ASATs is inevitable. Hence, since last two decades, *postprocessing of alarms*—processing the alarms after they are generated—is being explored as an alternative. The postprocessing goals include reducing the number of

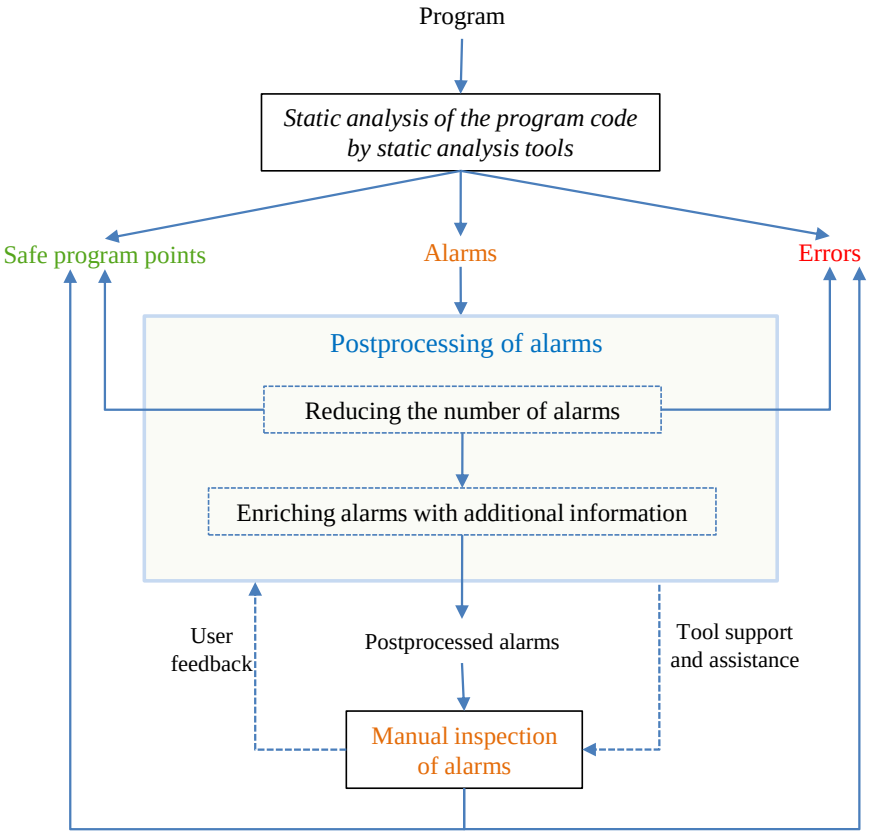


Figure 1.2: Illustrating postprocessing of alarms generated by static analysis tools.

alarms and effort required for their manual inspection. We use postprocessing of alarms to mean the following:

1. *Automatic processing of alarms* to reduce their number prior to reporting them to the user.
2. *Enriching alarms with additional information* so that effort to manually inspect them gets reduced.
3. *Simplifying manual inspection of alarms* by providing assistance and tool support during the inspection process.

In Figure 1.2 we present a generalized overview of postprocessing of alarms, covering all the cases listed above.

1.2.3 The Context of Our Work

The author has been working as a researcher in TRDDC, a research wing of Tata Consultancy Services[6], for more than 10 years. He is a member of research group that works *towards zero*

[6] https://www.tcs.com/

defects by designing new and scalable techniques for *verification and validation* of safety-critical systems. He has been working on improving usability of code proving tools used to detect common programming errors, primarily on safety-critical systems. Initially, the author was involved in design and development of a commercial tool, TCS Embedded Code Analyzer (TCS ECA) [197]. The tool helps analyzing safety-critical systems to detect common programming errors like *division by zero* and *array index out of bounds*, as well as to detect complex concurrency issues in multi-core architectures. The tool uses data flow analysis, semantic analysis, and abstract interpretation to analyze C and C++ applications. It supports detection of 25 types of defects, among which 15 are unique, i.e., not supported by any other static analysis tool, such as verification of implementation of sleep wakeup protocol [158]. The author has contributed to design and implementation of techniques for detecting defects of those unique types, and techniques for improving precision of underlying analyses such as value and pointer analyses. Later, to address the problem of alarms generated by TCS ECA, the author's focus shifted to postprocessing of alarms generated by a code proving tool [149, 150, 151, 152, 154]. The research work presented in this thesis also focuses on postprocessing of alarms.

1.3 Research Questions

Considering the benefits offered by postprocessing of alarms, a plentitude of techniques have been proposed [58, 84]. However, percentage of alarms remaining after the reduction range between 40% to 80% [36, 71, 133, 150, 223]. Moreover, even after simplification of manual inspection of alarms, the alarms require user's effort to inspect them manually. Therefore, we investigate how to improve existing postprocessing techniques. Hence, we ask the following central research question.

> **RQ:** How can we improve postprocessing of static analysis alarms?

In this thesis, we first survey the current state of postprocessing of alarms and identify limitations of the current techniques. We then focus on four of the identified limitations, propose techniques to overcome them, and empirically evaluate the techniques proposed. Figure 1.3 provides an overview of the limitations identified, corresponding research questions, and the chapters in which we answer them.

1.3.1 Understanding the Current State of Postprocessing of Alarms

While a plentitude of techniques have been proposed for postprocessing of alarms, their comprehensive overview is missing. In absence of such an overview,

- designers/developers and users of static analysis tools might spend considerable effort while they choose suitable postprocessing techniques from the plentitude of the existing ones; and
- researchers might be spending a large amount of effort in understanding the current state of alarms postprocessing, and they might be rediscovering existing approaches or miss opportunities to explore new directions.

Therefore, as a starting point for our work, we ask the following research question.

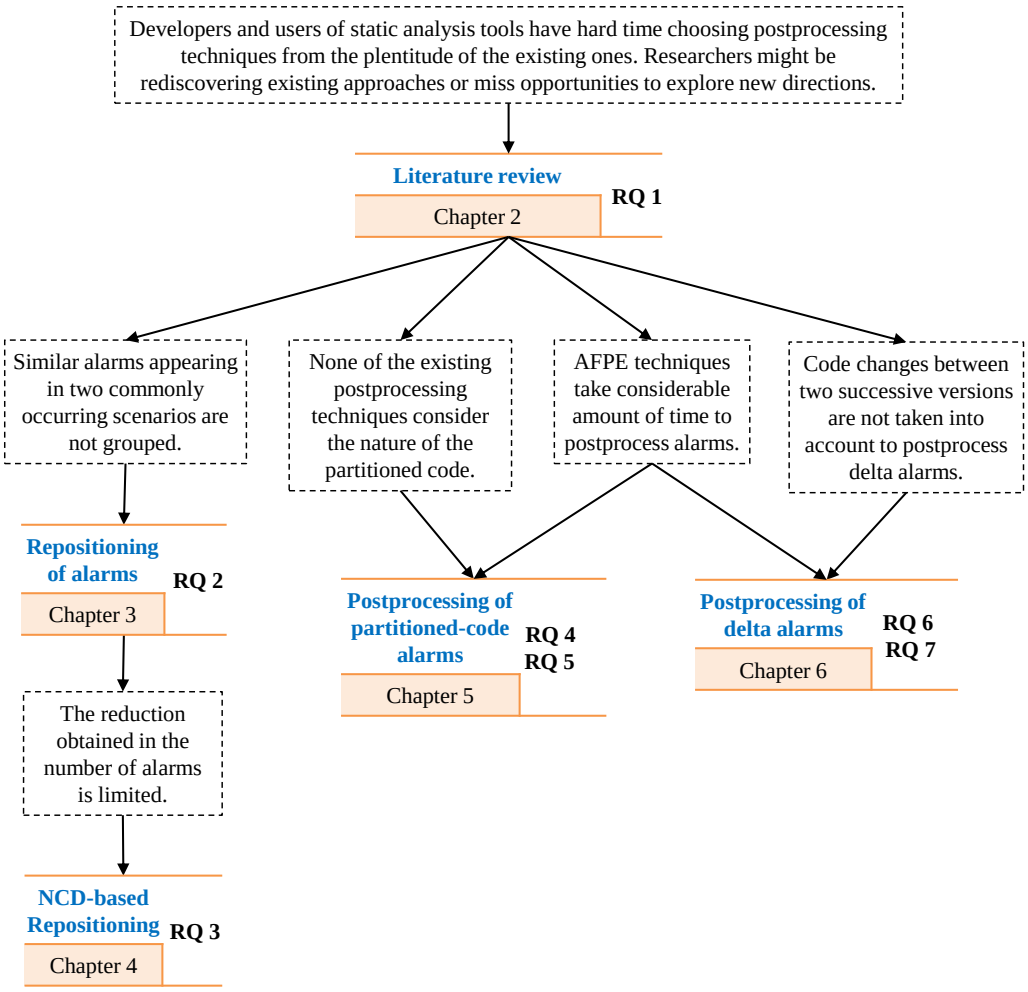


Figure 1.3: Thesis outline depicting the limitations and research questions addressed in the remaining chapters.

RQ 1: What approaches have been proposed for postprocessing of alarms?

To understand the current state of alarms postprocessing, we conduct a *systematic literature search* by combining *keywords-based database search* [138] and *snowballing* [15, 213]. We identify 130 primary studies that propose technique(s) for postprocessing of alarms, and identify six categories of approaches from them: clustering, ranking, pruning, automated false positives elimination (AFPE), combination of static and dynamic analyses, and simplification of manual inspection.

Recall that the main focus of our work is to improve postprocessing of alarms generated by a code proving ASAT (Section 1.2.3). Therefore, to identify areas of improvement, we studied

techniques implementing the approaches that are applicable for those alarms. Following we describe identified limitations of those techniques.

Our findings include that the approaches identified are complementary and they can be combined together in different ways.

Based on the literature study we identify the following limitations of existing techniques, and address them in this thesis.

Limitation 1 Clustering of alarms, one of the six categories of approaches that we identify, is commonly used to reduce the number of alarms. State-of-the-art clustering techniques [71, 150, 223] reduce the number of alarms by identifying a fewer dominant alarms for a group of similar/related alarms. *We have identified two commonly occurring scenarios in which the clustering techniques fail to group similar/related alarms.*

Limitation 2 In our earlier work [150, 151, 152, 153], we found that analyzing a partitioned-code results in generation of multiple alarms for the same POI but in the context of multiple partitions (Section 1.1.2.3). We call such alarms *common-POI alarms*. Since alarms generated on a partition are specific to that partition, the alarms are postprocessed partition-wise. The partition-wise postprocessing of common-POI alarms incurs redundant effort, e.g, manual inspection of the alarms or processing them using AFPE techniques results in analyzing the same code multiple times. However, none of the existing postprocessing techniques are explicitly directed to postprocess alarms generated on partitioned-code: the techniques do not consider the nature of the partitioned-code[7].

Limitation 3 Automated false positives elimination (AFPE), one of the six approaches that we identify, has gained popularity recently [34, 35, 152, 153, 205]. AFPE eliminates false positives by processing alarms using more precise techniques such as model checking and symbolic execution. The processing consists in generating assertions corresponding to alarms and then verifying the assertions using the precise techniques. We limit the discussion scope to *model checking-based* AFPE.

Existing model checking-based AFPE techniques require verifying a large number of assertions generated corresponding to the alarms [34, 152]. Moreover, the *context expansion* approach used to scale AFPE on very large systems increases the number of model checking calls made for a single assertion or a group of related assertions [34, 153]. Therefore, AFPE techniques suffer from poor performance: the evaluations of AFPE techniques [34, 152, 153] indicate that processing a group of similar/related assertions, on an average, involves making five calls to a model checker and it takes around four minutes. *Therefore, poor performance of AFPE techniques is a major concern when they are applied to alarms generated on very large systems.*

Limitation 4 Although software systems operating in the real world or modeling it, are evolving in nature, only a few techniques ([36, 133, 191, 207]) are proposed to postprocess alarms by considering the evolving nature of systems. Typically the techniques suppress alarms that repeat across two successive versions but are not impacted by code changes between the two versions, and report the remaining alarms (called *delta alarms*). *State-of-the-art postprocessing techniques take code changes into account only for computation of delta alarms but not for postprocessing those alarms further.*

[7]The only prior study that addressed grouping of alarms across partitions is our earlier publication [148] included in this thesis as Chapter 5.

1.3.2 Improving Clustering of Alarms

In our study to identify limitations of existing postprocessing techniques (Section 1.3.1), we observe that improving *clustering of alarms* can be expected to improve techniques implementing other postprocessing approaches as well. Moreover, alarms clustering techniques are generally less time-consuming as compared to the techniques implementing the other postprocessing approaches. Therefore, we choose to improve alarms clustering techniques. Based on our observed limitation of existing alarms clustering techniques (**Limitation 1**), we ask the following research question.

> **RQ 2:** How can we automatically group similar alarms that state-of-the-art alarms clustering techniques fail to group?

To overcome the limitation of existing clustering techniques, motivated by the work of Gehrke et al. [71] we propose repositioning of alarms. Repositioning reduces the number of alarms by moving groups of related alarms along the control flow to a program point where they can be replaced by a fewer ones. We evaluate proposed alarms repositioning on 16 open-source and four closed-source systems and observe that it reduces the number of alarms by up to 20% over state-of-the-art clustering techniques with median reduction of 7.25%.

While we expected our alarms repositioning to reduce the number of alarms significantly, the median reduction observed during the experimental evaluation is limited. This is why we take a closer look at reasons for the limited reduction in alarms. This led to the following research question.

> **RQ 3:** How can we improve the reduction in the number of alarms obtained by repositioning them?

To answer this question, we analyzed the cases where repositioning fails to group similar alarms which ideally should be grouped together. We find that, on average, 50% of the alarms resulting after their repositioning are similar, and 74% of the similar alarms are not grouped together due to a conservative assumption made about their immediate controlling conditions[8]. To further reduce the number of alarms, we introduce a notion of *non-impacting control dependencies* (NCDs) and propose a new variant of repositioning based on NCDs. We call the new variant *NCD-based repositioning*. We evaluate NCD-based repositioning on 16 open-source and 16 closed-source systems. The evaluation indicates that, compared to the original repositioning, NCD-based repositioning reduces the number of alarms by up to 36.09%, with median reduction being 10.48%.

1.3.3 Improving Postprocessing of Alarms Generated on Partitioned-code

Recall that presence of common-POI alarms increases the number of alarms generated on partitioned code (Section 1.1.2.3). Furthermore, the partition-wise postprocessing of common-POI

[8] A controlling condition of an alarm is a condition in a conditional statement that determines whether the alarm's program point will be reached.

alarms incurs redundancy (**Limitation 2**). However, none of the alarms postprocessing techniques address the problem of redundancy. Therefore, we turn our focus to improve postprocessing of common-POI alarms. We first targeted reducing redundancy in their manual inspection. To this end, we ask the following research question.

> **RQ 4:** How can we reduce redundancy in manual inspection of common-POI alarms?

To address the redundancy problem, we group together common-POI alarms and identify a set of functions for each group. The identified functions of a group are such that inspection of any one of the grouped alarms in the context of the identified functions guarantees the same result for other alarms in the same group when they are inspected in the context of the same functions. Based on these functions, we proposed a method to inspect a group of common-POI alarms. Our empirical evaluation results indicate that, on average, 45% of alarms generated on partitioned-code are common-POI alarms. The proposed method to inspect them manually reduces alarms inspection time by 60%.

We then turned our attention to improve efficiency of AFPE techniques applied to common-POI alarms (**Limitations 2 and 3**). We observed that, applying AFPE to common-POI alarms incurs redundancy similar to redundancy in their manual inspection. Reducing this redundancy allows to improve efficiency of AFPE techniques. To this end we ask the following research question.

> **RQ 5:** How can we reduce redundancy in AFPE applied to common-POI alarms?

To reduce the redundancy in AFPE applied to common-POI alarms and thus improve efficiency of AFPE, we reuse results of model checking calls across multiple partitions. Our empirical evaluation indicates that, the reuse of results improves efficiency of AFPE by up to 56%, with median improvement of 12.15%..

1.3.4 Improving Postprocessing of Alarms Generated on Evolving Code

Last, we revisit the previous observation that only a few alarms postprocessing techniques are proposed in the context of evolving code, and those techniques take the code changes into account only for computation of delta alarms but not to postprocess the alarms further (**Limitation 4**). Therefore, we aim at improving postprocessing of delta alarms generated in the context of evolving code.

In this work, we also revisited our finding that existing approaches and techniques for postprocessing of alarms are complementary, and thus can be combined together. Therefore, while addressing the above limitation, we also target combining different approaches identified for postprocessing of delta alarms. First, we focused to *rank* and *prune* delta alarms resulting after their clustering. Towards this we ask the following research question.

> **RQ 6:** How can we rank delta alarms based on types of the code changes generating them such that the alarms ranked higher are more likely to be errors than the alarms ranked lower?

To address this limitation, we analyze delta alarms generated on a few open source and industry applications. We observe that a high percentage of delta alarms gets falsely generated due to the conservative approach taken during their computation. Moreover, the code changes that generate delta alarms are of different types, and they impact their correspondingly generated delta alarms differently. Based on the type of code changes, we classify delta alarms into six classes, and then rank the alarms by prioritizing the classes. The ranking allows to suppress low priority alarms when time available for manual inspection is limited and the analysis purpose is bug finding. When the analysis purpose is code proving, similar to the existing alarms ranking techniques, our ranking technique helps to identify and fix errors early in the manual inspection. Our evaluation, based on delta alarms generated on 59 versions of seven open source applications, indicates that the proposed classification and ranking of delta alarms help to identify 61% of delta alarms as less likely to be errors than the others.

Next, we aim to address the problem of poor efficiency of AFPE applied to delta alarms. To this end, we ask the following research question.

RQ 7: How can we use code changes to improve efficiency of AFPE applied to delta alarms?

To improve efficiency of AFPE applied to delta alarms (**Limitation 3**), we adapt AFPE to take into account the classification of delta alarms and the code changes that generate them. In the adapted AFPE, based on the classification and code changes, we determine whether model checking results from the previous version can be reused during AFPE on the current version. The reuse of results allows to reduce the number of model checking calls, and thus to reduce the time taken by AFPE . Our empirical evaluation indicates that, the reuse of results across the versions reduces the number of model checking calls by median of 84.3%, which in turn reduces the AFPE time by 64.5%.

1.4 Thesis Outline and Origins of Chapters

As described above, this thesis contributes to the body of research on postprocessing of static analysis alarms by performing a study to understand its current state and proposing new alarms postprocessing techniques. Most of the work described in the remaining chapters has been published in peer-reviewed conferences. In Figure 1.3, we link each chapter (except Conclusions) to the research question(s) that it addresses and the study/postprocessing technique it describes. In the following, we briefly describe outline and origin of each chapter.

Chapter 2: Survey of Approaches for Postprocessing of Alarms In this chapter, we address RQ 1. We report on the systematic literature search that we performed by combining keywords-based database search and snowballing. The literature search was performed in June 2016 and later extended in January 2020. Based on the results of the literature search we identify six categories of approaches for postprocessing of alarms. We illustrate each category by discussing a few prominent techniques. Lastly, we discuss merits and demerits of each of the categories of approaches and our findings from this study. This chapter is based on and significantly extends the following publication.

[155] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 157–166, 2016. IEEE.

Chapter 3: Repositioning of Alarms In this chapter, we address RQ 2. We first informally discuss alarms repositioning as a means to overcome the limitation of state-of-the-art clustering techniques. Next, we present a data flow analysis-based technique to reposition alarms. Lastly, we evaluate the proposed repositioning technique by means of an empirical study. This chapter is based on the following publication.

[156] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of static analysis alarms. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 187–197, 2018. ACM.

Chapter 4: NCD-based Repositioning of Alarms In this chapter, we address RQ 3. We describe our intuition behind introducing the notion of *non-impacting control dependencies* (NCDs) of alarms. Since identifying NCDs is undecidable, next we discuss how their computation can be approximated. We then describe a novel technique to reposition alarms by taking into account the approximated NCDs. Last, we evaluate the technique by applying it to 16 open source and 16 closed-source systems. This chapter is based on the following publication.

[157] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Reducing static analysis alarms based on non-impacting control dependencies. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 115–135, 2019. Springer.

Chapter 5: Postprocessing of Alarms Generated on Partitioned Code In this chapter, we address RQ 4 and RQ 5. We describe our approach to group common-POI alarms, and present a method to inspect the grouped alarms. We then describe a reuse-based technique to improve efficiency of AFPE applied to common-POI alarms. The first part of this chapter, that answers RQ4, is based on the publication below, whereas a manuscript is under preparation based on the work that answers RQ 5.

[148] Tukaram Muske. Improving review of clustered-code analysis warnings. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 569–572. 2014. IEEE.

Chapter 6: Postprocessing of Delta Alarms In this chapter, we address RQ 6 and RQ 7. We begin by describing limitations of postprocessing techniques that compute *delta alarms*. Next, based on our observation that code changes of different types affect delta alarms differently, we design a technique to classify and rank the alarms. Later, we present a reuse-based technique to improve efficiency of AFPE. A manuscript is under preparation based on the work described in this chapter.

Chapter 7: Conclusions This final chapter concludes this thesis. It revisits the research questions and proposes directions for future research.

Suggested Method of Reading While each chapter has distinct contributions, there is some redundancy in sketching the context and providing details of the preliminaries required to describe presented techniques. This redundancy has not been eliminated in order to make it possible to read each chapter independently.

The presented techniques to perform/implement repositioning of alarms, in Chapters 3 and 4, are based on data flow analysis. The basic concepts related to data flow analysis are described in popular books by Nielson et al. [163] and Khedker et al. [100]. Assuming that reader is familiar with the basic concepts, we do not include them in preliminaries provided in those two chapters.

Wherever required we have revised our original work published in the conferences. The revision is to present this thesis in a coherent manner or reflecting a growing insight. Since evaluations of our proposed postprocessing techniques are on similar lines, we combine discussions of the threats to validity in *Conclusions* chapter (Section 7.3). Threats to validity, corresponding to the literature search, are discussed in the respective chapter (Chapter 2). Additionally, this thesis does not feature a separate chapter on related work. Rather, we discuss, on a per-chapter basis, the related work relevant to that chapter.

Chapter 2

Survey of Approaches for Postprocessing of Alarms

In this chapter, we review 130 primary studies that propose techniques for postprocessing of alarms. The studies are collected by combining keywords-based database search and snowballing. We categorize approaches proposed by the collected studies into six main categories. Furthermore, we categorize five of those categories into two or more sub-categories depending on methods and techniques used to implement the approaches. We provide an overview of the categories and sub-categories, their merits and shortcomings, and different techniques used to implement the approaches.

Since our work takes place in industry aiming at analysis of safety-critical systems (Section 1.2.3), we select and study (sub-)categories of the approaches that are useful when static analysis tools are used for code proving. *The findings of this study provide motivation for our work described in the subsequent chapters.*

2.1 Introduction

Automated static analysis tools (ASATs) have showcased their importance and usefulness in automated detection of code anomalies and defects. However, these tools generate a large number of alarms [20, 58, 92, 120]. Since last two decades, *postprocessing of alarms*—processing the alarms after they are generated by ASATs—is being explored as an alternative (Section 1.2.2). Recall, that we use postprocessing of alarms to mean the following:

1. *Automated processing of alarms* to reduce their number prior to reporting them to users.
2. *Enriching alarms with additional information* so that manual inspection effort gets reduced.
3. *Simplifying manual inspection of alarms* by providing assistance and tool support during the inspection process.

Note that the above described postprocessing of alarms does not consider reducing the number of alarms by making underlying static analysis more precise. That is, it excludes the option

of improving precision of underlying analyses, like value analysis and pointer analysis, implemented in ASATs. Moreover, in postprocessing of alarms, we do not differentiate between the two type of tools generating the alarms: code proving and bug finding tools.

Considering the benefits offered by postprocessing of alarms, a plentitude of approaches and techniques have been proposed [58, 84], and the approaches and techniques differ greatly. However, to the best of our knowledge, a comprehensive overview of these approaches and techniques is missing. In the absence of such an overview, (1) developers and users of static analysis tools have a hard time choosing postprocessing techniques from the plentitude of existing ones, and (2) researchers might be rediscovering existing approaches or miss opportunities to explore new directions. Therefore, in this chapter we ask the following research question.

RQ 1: What approaches have been proposed for postprocessing of alarms?

To understand the current state of alarms postprocessing, we performed a *systematic literature search* combining *keywords-based database search* [138] and *snowballing* [15, 213]. We combine the approaches to complement their strengths: the results of the former provided a start set required in the latter, and the latter identified the relevant papers which were missed by the former. The literature search was performed initially during the period of June 4 to June 14, 2016. During writing of this thesis, we extended the literature search to include the relevant studies that have been published after the initial search. We call the first search *initial literature search* and the extended one *extended literature search*.

Through the two literature searches, we identify 130 primary studies (research papers) that propose technique(s) for postprocessing of alarms, and identify six main categories of approaches from them. Furthermore, we categorize five of those categories into two or more sub-categories depending on methods and techniques used to implement the approaches. We provide an overview of the categories and sub-categories, their merits and shortcomings, and different techniques used in their implementations. Our findings include that the approaches identified are complementary and can be combined together in different ways.

Application of approaches from the identified (sub-)categories varies depending on purpose of ASAT generating the alarms: whether the ASAT is used for bug finding or code proving. Recall from Section 1.2.3 that the main focus of our work is improving ASATs used for code proving, e.g., to analyze safety-critical systems for their certification. Hence, we have zoomed in on (sub)-categories of the approaches that are applicable for alarms generated by code proving tools, and identified limitations of the techniques implementing those approaches (Section 2.6). We use our findings to improve the techniques in the next chapters.

The following are the contributions of this chapter.

1. A systematic literature search to identify studies that propose techniques for postprocessing of alarms.
2. Categorization of the approaches proposed for postprocessing of alarms into six main categories and further into sub-categories.
3. Study of the approaches and techniques to identify directions for further research.

Chapter Outline Section 2.2 describes the initial and extended literature searches. Section 2.3 summarizes data extracted from the relevant studies. Section 2.4 describes the identified

Table 2.1: Keywords used during the keywords-based database search.

I	1) elimination, 2) reduction, 3) simplification, 4) ranking, 5) classification, 6) reviewing, 7) inspection
II	1) static analysis, 2) automated code analysis, 3) source code analysis, 4) automated defects detection
III	1) alarm, 2) warning, 3) alert

categories of approaches for postprocessing of alarms, and Section 2.5 summarizes their merits and shortcomings. Section 2.6 describes our study performed to identify limitations of the techniques used for implementing the selected (sub)-categories of the approaches. Section 2.7 presents related work, and in Section 2.8 we present conclusions and discuss future work.

2.2 Systematic Literature Search

In this section, we present details of our literature search conducted to collect studies that propose techniques for postprocessing of alarms. Henceforth in this chapter, we use *studies* and *(research) papers* interchangeably. The literature search is performed by combining *keywords-based database search* [138] and *snowballing* [15, 213]. Performing a literature search is a time consuming activity. Moreover, it requires the searcher to be an expert in the area (postprocessing of alarms) and finding such a searcher is a hard task. Therefore, the literature search is performed by the author without involving additional experts. The author has 10 years of experience in developing a commercial static analysis tool (TCS ECA [197]) and research in designing new techniques for postprocessing of alarms (Section 1.2.3).

2.2.1 The Initial Literature Search

The initial literature search was conducted during the period of June 4, 2016 to June 14, 2016.

2.2.1.1 Keywords-based Database Search

Inspired by systematic literature reviews [108, 138], we conducted a keywords-based database search in Google Scholar[1] to collect the papers that propose techniques for postprocessing of alarms. We call these papers *relevant papers*. The keywords that we used during the search are listed in Table 2.1, and are identified from the research question RQ 1. The selection of keywords results in $84 = 7 \times 4 \times 3$ different search strings. A separate search is made in Google Scholar for each of the search strings, and we examined the first 150 results of every search. During this process, we examined a total of 12600 results[2]. For each paper in the results, we checked whether the paper is to be included in the collection of *relevant papers*. We identified a paper as *relevant* only if

- it proposes a technique, method, or an approach to postprocess alarms; and
- it is a peer-reviewed paper.

[1]Google Scholar. `https://scholar.google.com/`

[2]The number includes duplicates in the search results.

We excluded a paper from the collection of relevant papers if it deals with

- improving precision of the underlying static analyses like value analysis and pointer analysis, or refinements to the analyses (like [75, 95, 135]);
- an approach or methodology followed to reduce the number of alarms by designing lightweight static analysis tools;
- fault prediction or error/bug report triaging;
- mining of bug repositories in the context of software maintenance/evolution;
- study of economics or benefits of usage of static analysis tools (like [115, 228]); or
- evaluation, comparison, or benchmarking of precision of various static analysis tools (such as [30, 184, 198]).

While applying the inclusion/exclusion criteria for each paper, we considered the title, abstract, introduction/motivation, conclusion, and sometimes evaluation section of the paper. In the case of a paper satisfying both the inclusion and exclusion criteria, the paper was deemed to be relevant. This keywords-based search led to identification of 46 relevant papers.

2.2.1.2 Snowballing

After the keywords-based search, we performed snowballing [15, 213] due to the following reasons: (a) the search strings considered based on the keywords in Table 2.1, might be incomplete, e.g., due to terminological differences among the papers; and (b) more importantly, given a good *start set*, snowballing approach is found to be more effective and efficient in collecting relevant papers as compared to the keywords-based searches [15, 213]. By conducting snowballing after the keywords-based database search, we tried to identify and include as many relevant papers as possible, which were missed by the database search [119].

Creation of Start Set To begin with, a literature search using snowballing requires a start set having diversity in the included papers to avoid bias towards any specific class of papers and thus bias towards any specific approaches identified from them. Moreover, such a start set reduces the risk of missing a paper from clusters of papers not referring to each other [213]. In our literature search, we created the required start set by including all the relevant papers identified through the earlier keywords-based search. Thus, the start set used to perform snowballing included 46 relevant papers.

Backward and Forward Snowballing After the start set is created, we performed iterations of forward and backward snowballing. In the backward snowballing, the papers in *the reference list* of each relevant paper are examined to identify new papers to be included. In the forward snowballing, papers citing an included paper are examined to identify new relevant papers (citations analysis). We performed the citations analysis using Google Scholar. During the snowballing, we used the same inclusion/exclusion criteria that were used during the earlier keywords-based search.

In snowballing, iterations of the backward and forward snowballing are performed till saturation has been reached, i.e., until no new relevant papers are identified. In the snowballing search we conducted, two iterations of the backward and forward snowballing were sufficient.

During the snowballing process ca. 5800 papers[3] were examined for inclusion. With this search we identified 26 new relevant papers. This activity demonstrates that the combination of the two search approaches helped each approach to complement the other.

2.2.1.3 Relevant Papers Identified

Therefore, based on our initial literature search, performed by combining the two search approaches, we identified 72 papers that propose techniques for postprocessing of alarms.

In fact, the initial search has been published in the proceedings of SCAM 2016 [155]. This search was performed with a larger scope of *handling of alarms*, which included *designing of light-weight static analysis tools* also as an approach to handle alarms. This search identified 79 relevant papers. The number of papers relevant to *postprocessing of alarms*, 72, is identified after (1) excluding five papers that belonged to the additional approach (design of light-weight static analysis tools), and (2) excluding two relevant papers based on our improved understanding of postprocessing of alarms.

2.2.2 The Extended Literature Search

During writing of this thesis, we extended the literature search to include relevant papers that are published after the literature search, i.e., after June 14, 2016. The extended search is performed during the period of Dec 24, 2019 to Jan 10, 2020. To conduct the extended search, we followed the same methodology used to conduct the initial literature search.

2.2.2.1 Keywords-based Database Search

Using the same keywords in Table 2.1, we performed keywords-based search at Google Scholar. For each of the 84 search strings, we examined the papers that are published after the initial literature search, because the papers published before 2016 are already examined in the initial literature search. We used Google Scholar's filter option (Since 2016) to include in results only those papers that are published in 2016 or onwards. From the search results (i.e., papers that are published in 2016 or onwards), we considered and examined the first 150 results to identify relevant papers. During identification of the relevant papers, we used the same inclusion and exclusion criteria (Section 2.2.1.1). This search led to identification of 35 relevant papers.

2.2.2.2 Snowballing

We performed snowballing using 107 relevant papers as the required start set: 72 papers identified by the initial literature search, and 35 papers identified by the keywords-based search of the extended search. During the citations analysis (forward snowballing) we considered only the papers that are published in 2016 or onwards using the Google Scholar's filter option. During this search, two iterations of the forward and backward snowballing got performed, in which we examined 2345 papers[4]. This search led to identification of 23 additional relevant papers.

[3]The number includes duplicates in the search results.

[4]The number includes duplicates in the search results.

2.2.2.3 Relevant Papers Identified

The two literature searches, the initial and extended searches, together led to identification of 130 relevant papers. This number also includes three papers co-authored by us on which Chapters 3, 4, and 5 are based[5].

2.3 Data Extraction and Discussion

This section describes data extracted from those 130 relevant papers identified using the literature search (Section 2.2), followed by a few observations from the data extracted.

2.3.1 Data Extraction

We reviewed each of the relevant papers and extracted the following data: (1) the approach(es) proposed in the paper for postprocessing of alarms, (2) techniques and artifacts used to implement those approaches, (3) static analysis tools used for evaluating the approaches and techniques, and (4) Programming languages supported by the tools.

We used open tagging [194] to categorize approaches proposed by the papers. The tagging was performed by the author. The papers having similar approaches are grouped together, and a broader level approach is identified describing the group. When a paper is found to propose multiple approaches, i.e. the paper can belong to multiple categories, the most prominent approach mostly suggested by the title of the paper is selected to determine the category. For example, for the study by Kremenek et al. [113] that presents clustering and ranking of alarms and utilizes user-feedback for ranking purposes, we have identified ranking as its primary approach. Moreover, we categorized the identified categories further into sub-categories depending on the main characteristics of the approaches or techniques used to implement the approaches.

2.3.2 Discussion of Results

As a result of the above categorization, the following six categories of approaches are identified.

A. *Clustering: Alarms are clustered* into several groups based on similarity or correlations among them.

B. *Ranking: Alarms are ranked* using various characteristics of the alarms, the source code, history of bug/alarm fixes, code-commit history, and so on.

C. *Pruning:* Alarms are classified into two classes, actionable and non-actionable and *the non-actionable alarms are pruned*.

D. *Automated false positives elimination (AFPE):* Alarms are processed further using more precise techniques like model checking and symbolic execution *to automatically identify and eliminate false positives from the alarms*.

E. *Combination of static and dynamic analyses: Alarms are processed using dynamic analysis* to generate test cases that validate true errors.

[5]The paper [148], on which Chapter 5 is based, is identified through the initial literature search, whereas the papers [156] and [157], on which Chapters 3 and 4 are based respectively, are identified through the extended literature search.

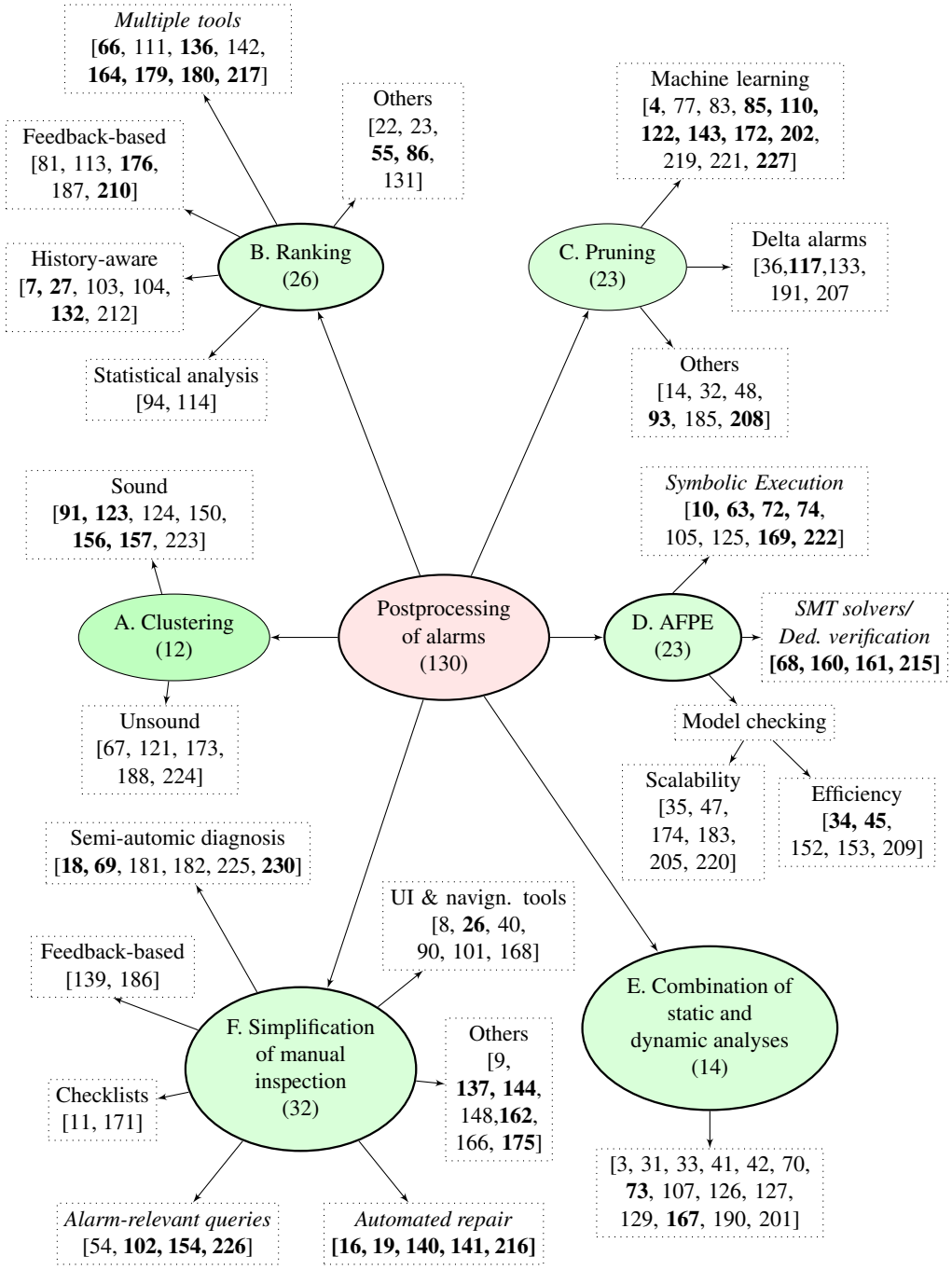


Figure 2.1: Summary of the (sub)-categories of approaches proposed for postprocessing of alarms and the corresponding relevant papers.

F. *Simplification of manual inspection: Manual inspection of alarms is simplified* by enriching alarms with additional information, providing tool support, and so on.

Figure 2.1 presents summary of the categorization of approaches for postprocessing of alarms, along with the number of papers in each category. Moreover, it presents the identified sub-categories and relevant papers belonging to them. In the figure, the sub-categories of approaches identified from relevant papers obtained through the initial (resp. extended) literature search are shown in non-italics (resp. *italics*). The relevant papers collected through the extended literature search are shown by marking them in bold. The presented categorization is described in detail in the next section (Section 2.4).

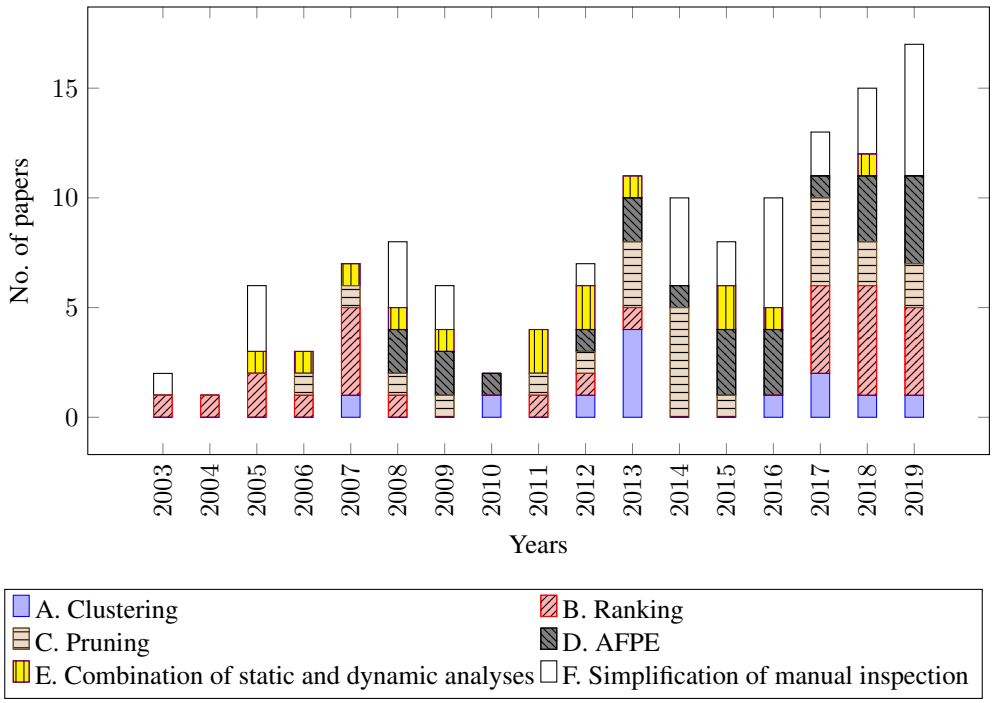


Figure 2.2: Number of the relevant papers published year- and category-wise.

Figure 2.2 presents year-wise distribution of the relevant papers per category of the approaches. It indicates that, there is continuous ongoing interest in the topic (postprocessing of alarms), and comparatively a higher number of papers are published recently (in the last three years). Moreover, simplification of manual inspection has been the more popular category comparatively, while ranking, pruning, and AFPE have received nearly equal popularity.

In Table 2.2 we summarize the data extracted from the relevant papers.

1. The (sub)-category of the approach identified for the paper (column *Cat.*).
2. The year of its publication (column *Year*).
3. Techniques and artifacts used to implement those approaches (columns *Techniques* and *Artifacts used* respectively).

Table 2.2: Summary of data extracted from the relevant papers collected through the initial and extended literature searches.

Categories	Sr. No.	Relevant papers	Srch	Year	Lang.	Tools	Techniques	Artifacts used
A. Clustering of alarms								
Sound	1	Jiao et al. [91]	E_K	2017	C	DTSC	feature functions	defect model
	2	Lee et al. [123]	E_K	2017	C	SPARROW	abstract interpretation	-
	3	Lee et al. [124]	I_K	2012	C	Airac	abstract interpretation, trace partitioning	-
	4	Muske et al. [150]	I_K	2013	C	TECA	data flow analysis	-
	5	Muske et al. [156]	E_K	2018	C	TCS ECA	data flow analysis	-
	6	Muske et al. [157]	E_K	2019	C	TCS ECA	data flow analysis	-
	7	Zhang et al. [223]	I_K	2013	C	DTSGCC	semantic slicing, error state slicing	-
Unsound	8	Fry et al. [67]	I_K	2013	C, Java	Coverity, FindBugs	graph theory	syntactic and structural info
	9	Le and Soffa [121]	I_K	2010	C	Phoenix	fault correlation graphs (graph theory)	modeled error states
	10	Podelski et al. [173]	I_K	2016	Java	Bucketeer	Craig interpolation	semantics-based signatures
	11	Sherriff et al. [188]	I_K	2007	C, C++	Matlab	singular value decomposition	alarm signatures
	12	Zhang et al. [224]	I_K	2013	C	DTSGCC	data mining	execution traces
B. Ranking of alarms								
Stat. analysis	13	Jung et al. [94]	I_K	2005	C	Airac	statistical analysis (Bayesian networks)	syntactic alarm contexts
	14	Kremenek and Engler [114]	I_S	2003	C	MC	statistical analysis (hypothesis testing)	number of alarms
History-aware	15	Aman et al. [7]	E_S	2019	Java	PMD	survival analysis	history of alarms
	16	Burhandenny et al. [27]	E_S	2017	Java	PMD	authorship of source files	history of alarms
	17	Kim and Ernst [103]	I_K	2007	Java	FindBugs, PMD, Jlint	statistical analysis	source code repository metrics
	18	Kim and Ernst [104]	I_K	2007	Java	FindBugs, PMD, Jlint	statistical analysis	source code repository metrics
	19	Liu et al. [132]	E_K	2018	Java	FindBugs	CNNs	alarm fix patterns
	20	Williams and Hollingsworth [212]	I_S	2005	Java	FindBugs	repository mining	bug repository metrics, alarm fix history
Feed-back -based	21	Heckman [81]	I_K	2007	Java	FindBugs	statistical analysis	alarm types, code locality, alarm fix hitory
	22	Kremenek et al. [113]	I_S	2004	C	MC	machine learning	code locality, user-feedback
	23	Raghothaman et al. [176]	E_K	2018	Java	Bingo	Bayesian inference	probabilistic model
	24	Shen et al. [187]	I_K	2011	Java	FindBugs	-	alarm patterns
	25	Wei et al. [210]	E_K	2017	Java	Android Lint	NLP techniques	user reviews
Multiple tools	26	Flynn et al. [66]	E_K	2018	C, C++, Java, Perl	SCALe	classification models	code base metrics, alarms fix history

Continued on next page

Table 2.2 – continued from previous page

Categories	Sr. No.	Relevant papers	Srch	Year	Lang.	Tools	Techniques	Artifacts used
	27	Kong et al. [111]	I_S	2007	C	RATS, ITS4, FLAW-FINDER	data fusion	alarm metrics
	28	Lu et al. [136]	E_K	2018	C/C++	Cppcheck, CBMC, Frama-C, Clang Static Analyzer	machine learning	defect types, code structures
	29	Meng et al. [142]	I_S	2008	Java	FindBugs, PMD, Jlint	policy prioritization	defect patterns
	30	Nunes et al. [164]	E_K	2019	PHP	Pixy, WAP, RIPS, phpSAFE, WeVerca	1-out-of-N strategy	(Non-) Vulnerable LOCs
	31	Ribeiro et al. [179]	E_K	2018	C, C++	Clang static analyzer, Cppcheck, Frama-C	ensemble learning	labeled alarms
	32	Ribeiro et al.[180]	E_S	2019	C, C++	Cppcheck, Frama-C, Clang Static Analyzer	ensemble learning	labeled alarms
	33	Xypolytos et al. [217]	E_K	2017	C	-	-	test suites
Others	34	Blackshear and Lahiri [22]	I_K	2013	C	ACSPEC	semantic reasoning	predicates and specifications
	35	Boogerd and Moonen [23]	I_K	2006	C	Codesurfer	graph theory	code metrics
	36	Nguyen Quang Do et al. [55]	E_K	2017	Java	CHEETAH	layered analysis	-
	37	Heo et al. [86]	E_S	2019	C	SPARROW	differential Bayesian inference	differential derivation graph, user feedback
	38	Liang et al. [131]	I_K	2012	Java	-	expressive defect pattern specification notation (EDPSN)	defect patterns
C. Pruning of alarms								
ML	39	Alikhashashneh et al. [4]	E_K	2018	C++	-	SVM, KNN, RIPPER Random forests	source code metrics
	40	Hanam et al. [77]	I_K	2014	Java	FindBugs	machine learning	alarm patterns
	41	Heckman and Williams [83]	I_S	2009	Java	-	machine learning	alarm characteristics
	42	Heo et al. [85]	E_K	2017	C	-	One-class SVM	codebase with bugs
	43	Koc et al. [110]	E_K	2017	Java	FindSecBugs	Bayes and LSTM models	code patterns
	44	Lee et al. [122]	E_K	2019	C,C++	-	CNNs	lexical patterns
	45	Meng et al. [143]	E_S	2017	C	-	machine learning	code property graph
	46	Pistoia et al. [172]	E_K	2017	Java	Phoenix, IBM Security AppScan Source	machine learning	syntactic properties of alarms

Continued on next page

Table 2.2 – continued from previous page

Categories	Sr. No.	Relevant papers	Srch	Year	Lang.	Tools	Techniques	Artifacts used
	47	Tripp et al. [202]	E_S	2014	Java-Script	-	machine learning	user feedback
	48	Yüksel and Sözer [221]	I_K	2013	C, C++	-	machine learning	alarm and code characteristics
	49	Yoon et al. [219]	I_K	2014	Java	SPARROW	machine learning	structural characteristics
	50	Zhang et al. [227]	E_K	2019	C	DTS	machine learning	characteristics of variables
Delta alarms	51	Chimdyalwar and Kumar [36]	I_K	2011	C	TECA	impact analysis	assertions
	52	Lahiri et al. [117]	E_S	2013	C	SymDiff	differential analysis	-
	53	Logozzo et al. [133]	I_K	2014	C#	cccheck	relative correctness	necessary/ sufficient conditions
	54	Spacco et al. [191]	I_S	2006	Java	FindBugs	fuzzier matching algorithms	warning signatures
	55	Venkatasubra-manyam and Gupta [207]	I_K	2014	C++	-	learning system	alarm and error patterns
Others	56	Ayewah et al. [14]	I_K	2007	Java	FindBugs	patterns identification	alarm types and patterns
	57	Chen et al. [32]	I_K	2013	C	RELAY	thread specialization	code regions
	58	Das et al. [48]	I_K	2015	C	Angelic-Verifier	abductive inference	angelic assertions, vocabulary
	59	Joshi et al. [93]	E_S	2012	C	CBUGS, POIROT	differential analysis	-
	60	Ruthruff et al. [185]	I_K	2008	Java	FindBugs	statistical models	code characteristics /metrics
	61	Wang et al. [208]	E_K	2018	C	Scan-build	-	fixed defects, critical functions
D. False positives elimination (FPE)								
Model checking - scala-bility	62	Chimdyalwar et al. [35]	I_S	2015	C	Polyspace, TCS ECA	BMC, loop abstraction	program slices
	63	Darke et al. [47]	I_K	2012	C	TECA, CBMC	BMC, loop abstraction	-
	64	Post et al. [174]	I_K	2008	C	Polyspace, CBMC, SATABS	BMC	-
	65	Rungta and Mercer [183]	I_S	2009	Java	Jlint, JPF	greedy depth first search	-
	66	Valdiviezo et al. [205]	I_K	2014	C++	Parfait, SPIN	model checking, program slicing	abstract programs, program slices
	67	Yu et al. [220]	I_S	2009	Java	-	fuzzy inference, model checking	code characteristics
Model checking - Effici-ency	68	Chimdyalwar and Darke [34]	E_S	2018	C	multiple tools	-	program slices
	69	Darke et al. [45]	E_S	2017	C	ELABMC, CBMC	loop abstraction, bounded model checking	program slices
	70	Muske et al. [152]	I_S	2013	C	TECA, CBMC	model checking	-

Continued on next page

Table 2.2 – continued from previous page

Categories	Sr. No.	Relevant papers	Srch	Year	Lang.	Tools	Techniques	Artifacts used
	71	Muske and Khedker [153]	I_K	2015	C	TCS ECA, CBMC	model checking, data flow analysis	-
	72	Wang et al. [209]	I_S	2008	C	Code-Auditor, BLAST	constraint-based analysis, model checking	constraints, program slices
Symbolic execution	73	Arzt et al. [10]	E_S	2015	Java	FlowDroid	symoblic execution	-
	74	Feist et al. [63]	E_S	2016	C	BINSEC/SE	dynamic symbolic execution	weighted slices
	75	Gerasimov [72]	E_K	2018	C	Svace, Anxiety	dynamic symbolic execution	-
	76	Gerasimov et al. [74]	E_S	2018	C	Avalanche	dynamic symbolic execution	-
	77	Kim et al. [105]	I_K	2010	C	Raccoon, Yices	abstract interpretation, symbolic execution	-
	78	Li et al. [125]	I_S	2013	C, C++	Flawfinder, SPLINT	trace analysis, symbolic execution	data flow tree
	79	Parvez et al. [169]	E_S	2016	C	WatSym, S2E, QEMU, KLEE	symoblic execution	-
	80	Zhang et al. [222]	E_K	2016	binaries	IDA pro, KLEE	dynamic symbolic execution	-
SMT/ Deductive verification	81	Gadelha et al. [68]	E_S	2019	C,C++	multiple tools	path-satisfiability analysis	-
	82	Nguyen et al. [160]	E_S	2019	C	Rose-checkers, Frama-C/WP, CBMC, Cobra	deductive verification, model checking, pattern matching	-
	83	Nguyen et al. [161]	E_K	2019	C	Rose-checkers, Frama-C/WP, CBMC, Cobra	deductive verification	-
	84	Xu et al. [215]	E_S	2019	C	LAID, Boolector	path-satisfiability analysis	-
E. Combination of static and dynamic analyses								
All	85	Aggarwal and Jalote [3]	I_S	2006	C	BOON, STOBO	-	-
	86	Chebaro et al. [31]	I_K	2012	C	Frama-C, Path-Crawler	program slicing	-
	87	Chen et al. [33]	I_S	2009	x86 binary	IntFinder	taint analysis	suspect instruction set
	88	Csallner et al. [41]	I_K	2005	Java	CnC	constraint solving	abstract/specific error conditions
	89	Csallner et al. [42]	I_K	2006	Java	DSD-Crasher	dynamic inference, dynamic verification	program invariants, test cases
	90	Ge et al. [70]	I_S	2011	C#	DyTa	dynamic test generation	-

Continued on next page

Table 2.2 – continued from previous page

Categories	Sr. No.	Relevant papers	Srch	Year	Lang.	Tools	Techniques	Artifacts used
	91	Gerasimov and Kruglov [73]	E_K	2018	C	Avalanche	dynamic analysis	-
	92	Kiss et al. [107]	I_K	2015	C, C++, Java	FLINDER-SCA	white-box fuzzing	-
	93	Li et al. [126]	I_K	2011	C,C++	Polyflow, Coverity, Clockwork	-	data flow graphs
	94	Li et al. [127]	I_S	2014	Java	RFBI	predictive dynamic analysis	programmer objections
	95	Li et al. [129]	I_K	2013	C, C++	HP Fortify	concolic testing	-
	96	Padmanabhuni and Tan [167]	E_K	2016	C	CodeSurfer, WEKA	dynamic analysis, machine learning	code characteristics
	97	Sözer [190]	I_K	2015	Java	FindBugs	Runtime monitoring	-
	98	Tomb et al. [201]	I_S	2007	Java	Check‘nŠ-Crash	symbolic execution	-
F. Simplification of manual inspection								
Semi-auto. diag-nosis	99	Barik et al. [18]	E_K	2016	Java	FIXBUGS	slow fixes	-
	100	Gao et al. [69]	E_K	2016	C	Fortify, KLEE	reachability	-
	101	Rival [181]	I_K	2005	-	Astrée	dependence analysis	abstract dependances
	102	Rival [182]	I_S	2005	C	Astrée	semantic slicing	alarm contexts
	103	Zhang and Myers [225]	I_S	2014	OCaml	-	expressive constraint language	constraints
	104	Zhu et al. [230]	E_K	2019	C	DTS	section-whole path generation strategy	inter-procedural diagnosis paths
Feed-back -based	105	Mangal et al. [139]	I_K	2015	Java	Eugene	probabilistic analysis	user-feedback
	106	Sadowski et al. [186]	I_S	2015	Multi-ple	Tricorder	data-driven ecosystem	user-feedback
Check-lists	107	Ayewah and Pugh [11]	I_K	2009	Java	FindBugs	systematic reviewing	review checklist
	108	Phang et al. [171]	I_S	2009	-	-	checklists-based review	triaging checklists
alarms-relevant queries	109	Dillig et al. [54]	I_K	2012	C	Compass, Mistral	abductive inference	alarm-specific queries
	110	Kim et al. [102]	E_K	2016	Java	Java Path Finder, FindBugs	symoblic execution	-
	111	Muske and Khedker [154]	E_K	2016	C	TCS ECA	data flow analysis	alarm root causes
	112	Zhang et al. [226]	E_K	2017	Java	URSA	integer linear programming	alarm root causes
Auto-mated repair	113	Bader et al. [16]	E_S	2019	Java	Infer, Error Prone	fix-patterns mining	alarms fix history
	114	Bavishi et al. [19]	E_S	2019	Java	FindBugs, PHOENIX	learning from examples	alarms fix history

Continued on next page

Table 2.2 – continued from previous page

Categories	Sr. No.	Relevant papers	Srch	Year	Lang.	Tools	Techniques	Artifacts used
	115	Marcilio et al. [140]	E_K	2019	Java	SPONGE-BUGS, SonarQube, SpotBugs	program transformation	alarms fixing templates
	116	Medeiros et al. [141]	E_S	2014	PHP	WAP, Pixy, PhpMinerII	machine learning	labeled alarms
	117	Xue et al. [216]	E_S	2019	Java	SonarQube	fix-patterns mining	alarms fix history
UI & navigation tools	118	Anderson et al. [8]	I_S	2003	C, C++	CodeSurfer	code navigation	program dependence graph
	119	Buckers et al. [26]	E_K	2017	Java	PMD, FindBugs, Checkstyle	user-interactive exploration	treemap code structure
	120	Cousot et al. [40]	I_K	2005	C	Astrée	code navigation	-
	121	Jetley et al. [90]	I_K	2008	C, C++	CodeSonar	code navigation	-
	122	Phang et al. [101]	I_K	2008	C	Locksmith	user interfaces	program paths
	123	Parnin et al. [168]	I_S	2008	Java	NOSE-PRINTS	code visualization	-
Others	124	Arai et al. [9]	I_S	2014	Java	GBC	gamification	points/scores
	125	Ma et al. [137]	E_S	2019				
	126	Menshchikov and Lepikhin [144]	E_K	2018	C,C++	multiple tools	report verbosity and generalization	-
	127	Muske [148]	I_K	2014	C	TCS ECA	review scope chains	call graphs
	128	Nguyen Quang Do et al. [162]	E_K	2018	-	-	video gaming principles	-
	129	Ostberg and Wagner [166]	I_S	2016	Java	HaST, FindBugs	salutogenesis model	various metrics, developer comments
	130	Querel and Rigby [175]	E_S	2018	Java	JLint, FindBugs	statistical models	code commits

4. Static analysis tools used for evaluating the approaches and techniques (column *Tools*).

5. Programming languages supported by the tools (column *Lang.*).

In the column *Srch* of Table 2.2, we denote the search method in which the corresponding paper is collected. We use I_K and I_S respectively to denote the keywords-based search and snowballing of the initial literature search, and E_K and E_S respectively to denote the keywords-based search and snowballing of the extended literature search.

Following are a few observations made from the data summarized in Table 2.2.

- ASATs that analyze *C* programs are used in 57% of the relevant studies for evaluating the postprocessing approaches and techniques proposed, i.e., *C* is the most popular programming language targeted by alarms postprocessing approaches. In total 65 different ASATs supporting analysis of *C* programs have been used.

- *Java* has been the second most popular language during evaluations in 36% of the relevant studies. In total these studies used 34 different ASATs to analyze Java programs. Among

these tools, FindBugs [13] has been found to be the most popular tool. PMD and Jlint have been other two commonly used ASATs for Java programs. This indicates that usage of shallow ASATs is common in analysis of Java programs.

2.3.3 Threats to Validity

Construct Validity Our literature study has focused on understanding what techniques have been proposed in scientific literature for postprocessing of alarms. Threats to construct validity concern how accurately we operationalize the notion of *scientific papers discussing postprocessing of alarms*, i.e. how we collect research papers that propose approaches for postprocessing of alarms.

In a keywords-based literature search, the selection of keywords (i.e., search strings used) affects the papers identified through the search [84, 108]. The keywords selected in our study are based on the author's experience in reading and writing research papers on the topic of postprocessing of alarms [150, 151, 152, 153, 154]. This list of search strings used may not be complete, and hence a relevant paper might be missed. Moreover, since reviewing all the results for each search string may not be practically possible, for each search string we reviewed only 150 search results. As a result, we might have missed relevant papers which appeared later in the results list. To mitigate these two issues, we performed snowballing by creating the start set from the relevant papers collected through the keywords-based search. The snowballing helped to identify relevant papers which were missed by the search.

Ideally, a literature search is to be performed by multiple researchers who have expertise in the topic under study: postprocessing of alarms. Getting such multiple experts is difficult because such experts are found to be rare, and the experts require to spend considerable amount of time in the search and reviewing the relevant papers. Therefore, the complete search, extraction of data, and categorization of approaches is performed solely by the author without involving additional experts. This might lead to subjective bias in the identification of relevant papers and data extraction. Moreover, the selected inclusion and exclusion criteria may introduce attrition bias [84].

Internal Validity Threats to internal validity concern the extent to which the observations are grounded in the data collected from the papers identified. Since the data extraction and categorization of the approaches are performed by the author only without involving additional experts, the categorization of the approaches (Figure 2.1) and the observations made from them (Table 2.3 and Section 2.5) might have been affected by the subjectivity bias. Comparing findings from our study with the similar studies is one way to validate the findings. However, no such studies exist. The existing studies about ASATs and alarms [49, 58, 84] have different goals and are not immediately comparable to our study.

External Validity Threats to external validity concern the extent to which results of the study generalize beyond the sample studied to the entire population. In our study, since the sample studied (i.e., 130 papers we have studied) is the same as the population (i.e., all papers ever published about postprocessing of alarms), threats to external validity are not applicable.

2.4 Details of Approaches for Postprocessing of Alarms

In this section, we describe the (sub)-categories of approaches (Figure 2.1), that we identified based on the relevant papers collected through the literature search. For each sub-category, we briefly describe a few relevant papers as its representatives.

2.4.1 Clustering of Alarms

In this category of the approaches, alarms are clustered (partitioned) into several groups based on similarity or correlation among them. Since alarms in a group are similar/correlated, generally only a few of them need to be inspected [123, 124, 150, 223], or all of them get inspected together [67, 121]. In both cases, clustering of alarms allows to reduce the overall inspection effort.

We categorize the approaches in this category further into *sound* and *unsound*. The categorization depends on whether the clustering approaches guarantee the following relationship among the alarms grouped together: *when one or more alarms in a cluster are false positives, all the other alarms in the same cluster are also false positives.* We call approaches that guarantee this relationship *sound*, otherwise *unsound*.

2.4.1.1 Sound Clustering

The approaches in this sub-category, cluster alarms by capturing the relationship among them [123, 124, 150, 223], i.e., when one or more alarms in a cluster are false positives, all the other alarms in the same cluster are also false positives. The one or more alarms in a cluster, identified based on the relationship, are called *dominant alarms* of the cluster. The implementation of these approaches is based on analysis techniques like data flow analysis [100] and abstract interpretation [38]. Due to the relationship between dominant and the other alarms in each cluster, inspection of the other alarms is not required when the dominant alarms are found to be false positives. This clustering approach is suitable for any ASAT since there are no false negatives arising due to skipping inspection of alarms other than the dominant alarms. This approach can be implemented in code proving tools used to verify safety-critical systems.

Our studies [156, 157], in which we have proposed *repositioning of alarms*, belong to this category of approaches. In these studies, we group more alarms together by overcoming limitations of the sound clustering techniques. The next two chapters (Chapters 3 and 4) are based on these studies.

2.4.1.2 Unsound Clustering

This sub-category relates to clustering alarms using similarity in syntactic or structural information, that is produced by static analysis tools or computed separately. This information relates to the code, alarm, or both. Unlike sound clustering, there are no guarantees on the relationship among alarms belonging to the same group. Due to this, skipping inspection of an alarm in a group can result in a false negative.

The techniques implementing this approach use heuristics to group similar alarms together, and propose to inspect those grouped alarms together. For example, Fry et al. [67] have used both structural and syntactic information to partition alarms into groups of related/similar alarms. The partitioning is based on the hypothesis that alarms on the same or similar execution paths may be related and can be inspected together to reduce inspection time. On similar lines, Podelski et

al. [173] have proposed a semantics-based signature for an alarm and the signatures are used to group the alarms.

Le and Soffa [121] have used *cause relationships* among the alarms—occurrence of one alarm can cause another alarm to occur—to group the alarms. They first construct a correlation graph by determining the error states of alarms and propagating the effects of the error states along the paths (cause relationships). Then they use the correlation graph to reduce the number of alarms that need to be inspected along a path. However, reducing the number of alarms this way may result in false negatives.

2.4.2 Ranking of Alarms

This category corresponds to prioritizing alarms such that the alarms that are more likely to be true errors are ordered up in the list. This approach is of help when not all alarms can be inspected and the inspection time should be spent in an effective way.

2.4.2.1 Statistical Analysis-based Ranking

The approaches in this sub-category are based on statistical analysis to rank alarms. For example, Kremenek and Engler [114] have employed a simple statistical model to rank alarms. It is based on the observation that, code containing many successful checks (safe cases analyzed by the tool) and a small number of alarms, tends to contain a real error. As another example, Jung et al. [94] have used a statistical method (Bayesian statistics) to compute probability of an alarm being true, and the probabilities are then used to rank the alarms.

2.4.2.2 History-aware Ranking

The approaches in this sub-category use history of alarm fixes as a basis to rank the alarms. For example, Kim and Ernst [103, 104] have ranked alarms by analyzing the software change history, where the categories of alarms that are quickly fixed by the programmers are treated as being more important. On similar lines, Aman et al. [7] estimate lifetimes of alarms by using survival analysis method, and assign higher priority to alarms which have shorter lifetimes: a shorter-life alarm is considered to be more important since many programmers resolved it sooner. Williams and Hollingsworth [212] have proposed a ranking scheme based on commonly fixed bugs and information automatically mined from the source code repository.

2.4.2.3 User Feedback-based Self-adaptive Ranking

In this sub-category, user feedback is used to rank alarms. For example, Shen et al. [187] have first assigned a predefined defect likelihood for each alarm pattern, and then ranked the alarms based on the defect likelihood. Later, the initial ranking is self-adaptively optimized based on feedback from users. On similar lines, Kremenek et al. [113] have used user-feedback to dynamically reorder ranked reports after each inspection. In their technique, Raghothaman et al. [176] first associate each alarm with a confidence value by performing Bayesian inference on a probabilistic model derived from the analysis rules. Later in subsequent iterations, user's feedback is captured during inspection of the alarms with the highest confidence, and the feedback is used to recompute the confidences of the remaining alarms. As another example, Heckman [81] has utilized user feedback from analyzed alarms, by combining it with alarm types and code locality, to rank the remaining alarms.

2.4.2.4 Multiple Tools Results-based Ranking

In this sub-category, results of multiple tools employing different static analysis methods are merged and ranked. In these studies, the merging of results enables results of different tools to validate each other, which in turn, greatly increases or decreases confidence about false positives and false negatives [66, 111, 164, 217]. In this sub-category we also include the studies that merge results of multiple ASATs to benefit from multiple and diverse ASATs [136, 142, 179, 180], and rank the merged alarms using other techniques. We have included them in this category as all those techniques deal with postprocessing of alarms generated by multiple ASATs.

Out of the eight primary studies in this sub-category, six studies are identified through the extended literature search. This indicates that the topic of combining results of multiple ASATs has recently (in the last three years) started attracting interest from the research community.

2.4.2.5 Other Techniques

A few other techniques used to rank alarms include the following:

- Static computation of execution likelihood of the program points at which alarms are reported, also has been used by Boogerd and Moonen [23] for ranking alarms.
- Liang et al. [131] have introduced a novel Expressive Defect Pattern Specification Notation (EDPSN) to define a resource-leak defect pattern more precisely, and have used it to prioritize resource leaks.
- Heo et al. [86] have proposed a technique to rank alarms generated on evolving code. They compute a graph that concisely and precisely captures differences between the derivations of alarms produced by an ASAT before and after the change. Later, they perform Bayesian inference, i.e., statistical inference in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available. The Bayesian inference is performed on the graph, which enables to rank alarms by likelihood of relevance to the change.

2.4.3 Pruning of Alarms

This category of approaches corresponds to classifying alarms into two classes, *actionable* and *non-actionable*. The classification of an alarm is based on the fact that alarms which are not acted upon by the user are seen as false positives and pruned, i.e., not reported to the user. Therefore, these approaches classify alarms depending on the likelihood of the user acting upon the alarms. As the pruned alarms are not guaranteed to be false positives, this approach can result in false negatives. This category is further organized based on the techniques employed to achieve pruning.

2.4.3.1 Machine Learning-based Pruning

Several studies, such as [4, 77, 83, 219, 221] have employed machine learning to differentiate between actionable and non-actionable alarms. For example, Hanam et al. [77] have achieved a binary classification by finding alarms with similar patterns, where the patterns are identified based on the code surrounding the alarms. Machine learning has been employed to account for semantic and syntactic differences during the identification of patterns. Yüksel and Sözer [221] have evaluated 34 machine learning algorithms in their study using 10 different artifact

characteristics. The plentitude of studies that use machine learning to prune alarms [4, 77, 83, 85, 110, 122, 143, 172, 202, 219, 221, 227] suggests popularity of the approach among researchers.

2.4.3.2 Computation of Delta Alarms

The approaches in this sub-category reduce alarms generated during analysis of evolving software by pruning alarms that repeat across versions. Techniques employing this approach apply various analyses to identify (1) alarms that are newly generated as compared to alarms on the previous code version, and (2) repeated alarms which are impacted by the code changes. Alarms reported after applying these techniques are called *delta alarms*.

The techniques proposed to compute delta alarms [36, 117, 133, 191, 207] vary in the methods they use for computation. For example, Spacco et al. [191] have identified newly generated alarms as compared to the previous version, by matching alarms through two approaches: *pairing* and *alarm signatures*. Chimdyalwar and Kumar [36] have proposed an approach to prune repeated alarms generated on evolving software systems. The pruning is achieved by performing an impact analysis—analyzing the impact of changes made between the two successive versions on the alarms—and suppressing alarms that are not impacted by the changes.

Logozzo et al. [133] have introduced a new static analysis technique (Verification Modulo Versions) for reducing the number of alarms while providing sound semantic guarantees. The proposed technique first extracts semantic environment conditions—sufficient or necessary conditions—from a base program (previous version) and uses those conditions to instrument a new version. Later, the instrumented code is verified, which results in pruning of alarms.

2.4.3.3 Other Techniques

Following are a few other techniques proposed to prune alarms.

- Statistical models are used by Ruthruff et al. [185] to identify and prune non-actionable alarms.
- Das et al. [48] have constrained the analysis verifier to report alarms only when no acceptable environment specification (specified through a vocabulary) exists to prove the assertion.
- Chen et al. [32] have pruned alarms corresponding to data-races through thread specialization: distinguishing the threads statically by assigning IDs to threads and fixing their number.

2.4.4 Automated False Positives Elimination

In this approach, more precise techniques like model checking and symbolic execution are used to identify and eliminate false positives from alarms. An assertion is generated corresponding to each alarm and it is verified using model checkers [35, 47, 174, 205, 220] or tools based on symbolic execution [10, 63, 72, 74, 105, 125]. The approaches in this category are more precise as compared to the other approaches, as they *precisely eliminate* false positives from alarms without any user intervention (inspection). However, the postprocessing of alarms in this approach generally faces the issues of non-scalability and poor performance due to the state space problem associated with model checking.

We organize this category into sub-categories based on the techniques used in AFPE: model checking, symbolic execution, and deductive verification.

2.4.4.1 Model Checking-based AFPE

In this approach, model checkers such as CBMC [28] are used to eliminate false positives. We partition the approaches in this sub-category into two parts depending on whether they address the *non-scalability* or *poor-efficiency* issues related to the AFPE process.

Note that other combinations of static analysis and model checking have been proposed in the literature [25, 62, 95], where these two techniques iteratively exchange information. We treat this approach differently from false positives elimination (postprocessing of alarms), because the aim of this combination is to improve precision of static analysis and improving the analysis precision is out of scope of postprocessing of alarms (Section 2.2.1.1).

Achieving Scalability Post et al. [174] have proposed an incremental approach, called *context expansion*, to use a model checker in a more scalable way. In this approach, verification of assertion(s) starts from the function containing the assertions, and then the verification context is gradually incremented to the direct and indirect callers of the function. That is, the verification is started with the minimal context and the context is expanded later on a need basis. This approach also has been observed to be beneficial by other studies [47, 153].

Program slicing [199] also has been another commonly used technique for reducing the state space, and in turn, achieving scalability [35, 47]. On similar lines, a notion of abstract programs has been proposed by Valdiviezo et al. [205] to achieve scalability of model checkers.

Improving Efficiency The model checking-based AFPE has been found to have poor efficiency due to (1) a large number of alarms that need to be processed, (2) multiple model checking calls for a single assertion due to the context expansion [174], and (3) considerable amount of time (on an average 3 to 5 minutes[6]) that model checking usually takes. To address this issue, i.e., to improve efficiency of AFPE, different techniques have been proposed. For example, Muske et al. [152, 153] have proposed static analysis-based techniques to predict outcome of a given model checking call. The predictions are used to reduce the number of model checking calls and thus, improve AFPE efficiency. Darke et al. [34] partition the generated assertions into disjoint groups based on the data and control flow characteristics, and verify assertions in one group at a time. Wang et al. [209] have used program slicing to improve efficiency of model checking-based AFPE.

2.4.4.2 Symbolic Execution-based AFPE

In this sub-category, symbolic execution is used to eliminate false positives [105, 125, 222]. In symbolic execution, instead of supplying the concrete inputs to a program (e.g. numbers), symbols representing arbitrary values are supplied, and the values of program variables are represented with symbolic expressions [106]. To address the issue of too many execution paths during symbolic execution, several studies use a variant of symbolic execution, called dynamic symbolic execution (or concolic execution) [10, 63, 72, 74, 169]. This variant involves running a symbolic execution along with a concrete one.

2.4.4.3 SMT Solvers/Deductive Verification-based AFPE

In this sub-category, SMT solvers [50, 147] or deductive verification (also called *theorem proving*) [65] are used to eliminate false positives [68, 160, 161, 215]. For example, Nguyen et al.

[6]This time taken includes the time taken to generate program slices before the assertion is verified by a model checker.

[160, 161] use deductive verification to eliminate false positives. In the studies that use SMT solvers for AFPE [68, 215], for each alarm, constraints are generated that represent the conditions under which the alarm is an error. Then, the constraints are checked using a SMT solver to determine their satisfiability. When the constraints are found to be unsatisfiable, the alarm is identified as a false positive and eliminated.

2.4.5 Combination of Static and Dynamic Analyses

As a general theme of approaches in this category, static analysis alarms are checked using dynamic analysis if they are true errors and the test cases witnessing failures are reported as error scenarios to the user. This combination requires executing the programs, which is usually absent in static analysis. A few of these studies adopting this combination approach are described below.

Csallner et al. [41] have combined static analysis and concrete test-case generation (Check-n-Crash tool), where a constraint solver is used to derive specific instances of abstract error conditions identified by a static checker (ESC/Java). Later, actual test cases exhibiting error scenarios uncovered by true alarms are presented to the users. As an advancement to this approach, Csallner et al. [42] have used a three step approach, consisting of dynamic inference, static analysis, and dynamic verification (DSD-Crasher tool). The processing in the approach includes (a) inferring likely program invariants using dynamic analysis, (b) using the invariants as assumptions during the static analysis step, and (c) generating test cases that validate true alarms.

Program slicing also has been used for the efficiency of techniques employing this approach: confirming/rejecting more number of alarms in a given time [31]. The efficiency is achieved by reporting more precise error information on simpler programs having shorter program paths and showing values for useful variables only. This reporting reduces the alarms analysis and correction time by the tool users.

Li et al. [127] have proposed a concept of residual investigation—a dynamic analysis serving as the runtime agent of a static analysis—for checking if an alarm is likely to be true. The novelty of the proposed approach lies in predicting errors in executions, which are not actually observed. This predictive nature of their approach is of significant advantage when generation of test cases is hard for very large and complex programs [127].

2.4.6 Simplification of Manual Inspection

The approaches in this category aim to simplify manual inspection of alarms by supporting users during the inspection process. Based on the methods used for the simplification, we organize the approaches into the seven sub-categories described below.

2.4.6.1 Semi-automatic Alarm Inspection

This sub-category relates to providing support for semi-automatic inspection of alarms. For example, to help the user in inspection of alarms by making the inspection more automatic, Rival [181] has enhanced semantic slicing (i.e., computation of precise abstract invariants for a set of erroneous traces) with information about abstract dependences. An abstract dependence is a dependence that can be observed by looking at abstractions of the values of the variables only. In another study, Rival [182] has proposed a framework for semi-automatic inspection of alarms. In this framework, an initial static analysis is refined into an approximation of a subset of traces that actually lead to an error. Later, a combination of forward and backward analyses is used to

prove whether this set is empty. If this set is proved to be empty, the alarm is concluded as a false positive.

As another example, Zhu et al. [230] have proposed a novel approach that combines demand-driven analysis and inter-procedural data flow analysis. Using the combined analyses inter-procedural paths are generated to help the user inspect alarms automatically.

2.4.6.2 Feedback-based Manual Inspection

A few studies have been found to capture user-feedback to simplify manual inspection of alarms. For example, Mangal et al. [139] have formulated user-guided program analysis to shift decisions about the kind and degree of approximations to apply in an analysis from the analysis writer to the analysis user. In the proposed analysis approach, user feedback about which analysis results are liked or disliked is captured and the analysis is re-run [139]. This approach uses soft rules to capture the user preferences and allows users to control both the precision and scalability of the analysis.

Sadowski et al. [186] have proposed a program analysis platform to build a data-driven ecosystem around static analysis. The platform is based on a feedback loop between the users of static analysis tool(s) and writers of those tool(s). The feedback loop is towards simplifying inspection of alarms reported by the tools.

2.4.6.3 Checklists-based Manual Inspection

In this approach, checklists are used to systematically guide users during manual inspection of alarms. Ayewah et al. [11] have proposed use of checklists to enable more detailed review of static analysis alarms. On similar lines, Phang et al. [171] have used triaging checklists to provide systematic guidance to users during manual inspection of alarms. The users follow the instructions on the checklist, during manual inspection of alarms, to answer each question and to determine conclusions about the alarms. It also proposes that the checklists are designed by tool developers so that, (a) known sources of imprecision in their tools are pointed out, and (b) users are instructed on how to look for those sources of imprecision. Additionally, the checklists are customized to individual alarms so that a minimum number of questions are answered during inspection of an alarm.

2.4.6.4 Alarms-relevant Queries

In this approach, alarm-specific queries are presented to the user for achieving effective and efficient manual inspection of alarms. For example, Dillig et al. [54] have proposed an approach to classify alarms semi-automatically as errors or non-errors by presenting alarm-specific queries to the users. Abductive inference is used to compute small and relevant queries that capture exactly the information needed from user to discharge or validate an error. Two types of queries (proof obligation and failure witness queries) are framed, and they are ranked using a cost function so that easy-to-answer queries are presented first to the users.

Zhang et al. [226] have combined a sound but imprecise analysis with precise but unsound heuristics, through user interaction. This combined approach poses questions to the user about the root causes that are targeted by the heuristic. If the user confirms them, only then is the heuristic applied to eliminate the false alarms. Muske and Khedker [154] have proposed cause points analysis. In this analysis, root causes of alarms are identified, and queries generated specific to the causes are presented to the user for reducing the manual inspection effort.

2.4.6.5 Automated Repair

Recent studies have also been aiming at automated repair of alarms. For example, Bavishi et al. [19] have proposed a technique for automatically generating patches for static analysis violations by learning from examples. Aiming at improving usability of ASATs, Marcilio et al. [140] automatically provide fix suggestions, that is modifications to the source code that make it compliant with the rules checked by the ASATs. As another example of this approach, Xue et al. [216] propose a history-driven approach to automatically fix code quality issues detected by ASATs, by utilizing the fixing knowledge mined from the change history in the code repositories.

2.4.6.6 Usage of Novel User-interfaces/Visualization Tools

This sub-category deals with usage of code navigation/visualization tools that simplify the code traversals performed by the user while inspecting alarms manually [8, 40, 90]. For example, Phang et al. [101] have presented a novel user interface toolkit (path projection) to help users to visualize, navigate, and understand program paths during the inspection. Parnin et al. [168] have used a catalogue of lightweight visualizations to help users in reviewing the alarms. In their study, a simple light-weight visualization is designed for each alarm. Buckers et al. [26] have proposed UAV (Unified ASAT Visualizer) that provides an intuitive visualization, enabling developers, researchers, and tool creators to compare the complementary strengths and overlaps of different Java ASATs. The UAV's enriched treemap and source code views provide its users with a seamless exploration of the alarm distribution from a high-level overview down to the source code.

2.4.6.7 Other Techniques

Following are a few other techniques proposed to simplify inspection of alarms.

- Arai et al. [9] have explored a gamification approach and proposed a novel gamified tool for motivating developers to remove alarms through manual inspection. The tool proposed calculates scores based on the alarms removed (inspected) by each developer or team. On similar lines, Nguyen Quang Do et al. [162] have proposed to leverage the knowledge of game designers, and to integrate gaming elements into analysis tools to improve their user experience.
- In our study [148], on which Chapter 5 is based, targeting reduction in the inspection effort, we have proposed a method to group and inspect alarms that are reported for the same POI but belonging to different code-partitions (Section 1.1.2.3). The proposed method allows to identify alarms in most of the groups as false positives by inspecting only one alarm from those groups.
- The overflow of information and decisions to be made during manual inspection of alarms can be tiring and cause stress symptoms to the reviewers. To fight stress during the inspection, Ostberg and Wagner [166] have proposed to use *salutogenesis model* that has been used in health care.
- Menshchikov and Lepikhin [144], aiming at evaluating and improving quality of reports of ASATs, generalize the tool output messages and explore ways to improve reliability of comparison results. To this end, they introduce informational value as a measure of report quality with respect to 5Ws (What, When, Where, Who, Why) and 1H (How To Fix) questions.

No.	Category	Merits	Shortcomings
A.	Clustering	In sound clustering, mostly dominant alarms need to be inspected while skipping inspection of the other alarms, whereas in unsound clustering, the group-wise inspection of alarms reduces inspection effort.	Unsound clustering may result in false negatives. Reduction in the number of alarms (by sound clustering) depends on the percentage of alarms identified as dominant.
B.	Ranking	During manual inspection of alarms, the alarms that are more likely to be errors get inspected early, and this approach does not result in false negatives.	It requires inspecting all the reported alarms.
C.	Pruning	Only the actionable alarms are to be inspected, while the other alarms being identified as non-actionable are not reported to the user.	This pruning approach may result in false negatives as a pruned alarm cannot be guaranteed to be a false positive (except the reliable techniques that compute delta alarms [36, 133]).
D.	False positives elimination	It is automatic and more precise as compared to the other approaches to reduce the number of alarms.	Techniques employing this approach usually face issues related to non-scalability and poor efficiency.
E.	Combination of static and dynamic analyses	It presents error scenarios for true alarms.	It requires support for executing the programs (e.g., test cases, and runtime environment) which is usually absent during static analysis process.
F.	Simplification of manual inspection	It provides significant aid to the users during manual inspection.	User involvement is a must.

Table 2.3: Merits and shortcomings of the categories of approaches for postprocessing of alarms.

2.5 Discussion

Table 2.3 summarizes merits and shortcomings of the identified approaches for postprocessing of alarms. It shows that the categories of approaches are complementary: shortcomings of one approach are merits of some other approaches and they can complement each other.

We observe that possible combinations of the approaches are not widely studied or evaluated. There exist a few studies, such as [113, 152], that consider the combinations of the approaches and find them to be promising. Thus, exploring the possible combinations of these various approaches can be a future research direction to postprocess alarms more effectively. For example, only the dominant alarms identified through sound clustering (Section 2.4.1.1) can be ranked (Section 2.4.2) or pruned (Section 2.4.3). In another example, pruning of alarms followed by AFPE (Section 2.4.4) can help each other: AFPE eliminates false positives from the actionable alarms resulting after pruning, and processing only the actionable alarms (a subset of alarms generated) reduces the number of alarms to be processed by the AFPE techniques.

Moreover, combinations of the approaches can be implemented with two strategies: sequencing the approaches one after the other (pipelining), and running them in parallel. The positioning of approaches in the combinations with pipelining can vary depending on the requirements in practice. For example, choice for the first approach to be implemented, between pruning and AFPE when they are to be combined, can be made based on total time available for processing the alarms. The other strategy, parallelization of the approaches can help in enhancing confidence about false positives and true errors. For example, the results obtained in isolation from approaches like ranking, pruning, and combination of static and dynamic analyses, can be merged together to increase confidence about alarms that are more likely to be false positives or errors.

The above combinations of the approaches also may introduce a performance penalty. Thus, performance of such combinations also needs to be studied while studying advantages and disadvantages of the combinations. We believe, given the high computing power of machines commonly available today, implementing such combinations in practice is feasible and can help practitioners considerably.

Among the identified six main categories, (sound) clustering, ranking, AFPE, and simplification of manual inspection, do not result in false negatives[7]. Thus, these approaches can be implemented in code proving ASATs. All categories and sub-categories can help to postprocess alarms generated by a bug finding ASAT.

2.6 Detailed Study of the Approaches

In this section, we describe our study of alarms postprocessing techniques to identify their limitations. The work presented in the next chapters overcomes the identified limitations.

Recall from Section 1.2.3 that the main focus of our work is improving ASATs used for code proving, e.g., to analyze safety-critical systems for their certification. In particular, we aim to improve postprocessing of alarms generated by our commercial static analysis tool, TCS ECA [197]. Therefore, we select the following four (sub)-categories of the approaches that are applicable in the context of code proving: (1) sound clustering, (2) computation of delta alarms (pruning), (3) AFPE, and (4) simplification of manual inspection. We then studied the techniques implementing the selected approaches to identify areas of improvement. Following we describe identified limitations of those techniques.

Sound Clustering of Alarms The approaches in this sub-category are proposed to reduce the number of alarms by identifying fewer dominant alarms for a group of similar/related alarms. We find that state-of-the-art techniques [123, 150, 223] proposed to implement sound clustering of alarms fail to group similar alarms that appear in commonly occurring scenarios, e.g., when similar alarms appear in the different branches of an *if* condition. The failure is due to the conservative assumption these techniques make about the transitive control dependencies of the alarms. Therefore, improving these techniques to group alarms in those scenarios can help to further reduce the number of alarms by the clustering approach. Our work presented in Chapters 3 and 4 is motivated by this observation.

Simplification of Manual Inspection TCS ECA employs code partitioning approach to address the issues like incomplete code and non-scalability on very large systems. As discussed

[7]No alarm is removed/pruned unless it is guaranteed be false either automatically or by the tool-user.

in Section 1.1.2.3, multiple alarms can get generated for a single program point when the point belongs to multiple parts (called *common-POI alarms*). Therefore, more alarms get generated on partitioned-code compared to the corresponding non-partitioned code. We find that

1. postprocessing common-POI alarms in the context of each of their code partitions incurs redundant effort related to, e.g, manual inspection of those alarms or processing them using AFPE techniques; and

2. although several techniques are proposed for postprocessing of alarms, none of the techniques considers the nature of partitioned-code during postprocessing of common-POI alarms[8], and therefore the techniques are unable to eliminate the redundancy in postprocessing of common-POI alarms.

Considering the nature of partitioned-code during postprocessing of common-POI alarms can help to reduce the redundant effort incurred during the postprocessing. Our work presented in Chapter 5 is motivated by this observation.

Computation of Delta Alarms We find that, state-of-the-art techniques [36, 133, 191, 207] that compute delta alarms take code changes into account only for computation of delta alarms but not to postprocess those alarms further. Postprocessing the delta alarms further based on the code changes can help obtaining further benefits. The work presented in Chapter 6 is motivated by this observation.

Automated False Positives Elimination Based on evaluations of the techniques proposed to improve efficiency of AFPE [34, 152, 153], we find that even after applying those techniques AFPE suffers from poor efficiency: the evaluations of the AFPE techniques [34, 152, 153] indicate that processing a group of similar/related assertions, on an average, involves making five calls to a model checker and it takes around four minutes. Therefore, poor performance of AFPE techniques is a major concern when they are applied to very large systems. This observation led to our work presented in Chapters 5 and 6.

2.7 Related Work

In this section, we compare our work with recently published (1) literature reviews of techniques for postprocessing of alarms, and (2) studies about evaluation or benchmarking of tools/techniques that postprocess alarms. We start by comparing our literature search with the systematic literature review conducted by Heckman and Williams [84] as it reviews techniques for ranking and pruning of alarms. In this review, approaches proposed by 21 different studies for postprocessing of alarms, are analyzed and categorized into two categories, ranking and pruning. Among those 21 studies, 10 studies propose classification of alarms into actionable or non-actionable classes, while the other 11 studies propose ranking of alarms. Compared to this review, our literature search is more comprehensive as it includes more studies (130) that propose a variety of approaches for alarms postprocessing. For example, due to inclusion of those additional studies, we could identify new categories like clustering, automated false positives elimination, and simplification of manual inspection. Moreover, wherever suited, the studies in each category

[8]The only prior study that addressed grouping of alarms across partitions is our earlier study [148] included in this thesis as Chapter 5.

are further categorized into multiple sub-categories. This categorization helps to understand the proposed approaches better and comprehensively. Five of those 21 studies included in the review by Heckman and Williams [84] were not included in our study: the excluded studies performed evaluations of ASATs (e.g. [82] and [165]) rather than introducing new postprocessing techniques.

In their mapping study, Mendonca et al. [49] have selected and analyzed 51 studies to identify state-of-the-art static analysis techniques and tools, and main approaches developed for postprocessing of alarms. In our study, as our focus was on different approaches through which alarms are postprocessed, we did not include 39 of those 51 studies. The excluded studies dealt with improving analysis precision, study of defects, and even evaluation and benchmarking of ASATs.

In their mapping study, Elberzhager et al. [58] have classified and provided analysis of approaches that combine static analysis and dynamic quality assurance techniques. The static quality assurance techniques deal with code reviews, inspections, walkthroughs, and usage of static analysis tools, whereas our literature search is with much broader scope of postprocessing of alarms: in our survey, combination of static and dynamic analyses is one of the categories of approaches to postprocess alarms.

Li et al. [128] have performed a systematic literature review to provide overview of state-of-the-art works that statically analyze Android apps. From these works, they highlight the trends of static analysis approaches, pinpoint where the focus has been put, and enumerate the key aspects where future research is still needed. In this review, 124 research papers are studied. The review is performed with much broader scope: the review is performed mainly in the following five dimensions (1) problems targeted by the approach, (2) fundamental techniques used by authors, (3) static analysis sensitivities considered, (4) Android characteristics taken into account, and (5) the scale of evaluation performed. Unlike to this review, our study is focused on understanding the state of postprocessing of alarms irrespective of domains of the applications being analyzed.

To the best our knowledge, there does not exist other literature survey or reviews studying alarms postprocessing approaches. Several other studies, like [6, 130, 218] evaluate various techniques for alarms postprocessing. Allier et al. [6] have proposed a framework for comparing different alarms ranking techniques and identifying the best approach to rank alarms. The various techniques proposed for postprocessing of alarms are compared using a benchmark having programs in Java and Smalltalk, and three static analysis tools: FindBugs, PMD, and SmallLint. Using this framework, algorithms to rank alarms are compared. In another study, Liang et al. [130] have proposed an approach for constructing a training set automatically, required for effectively computing the learning weights for different impact factors. These weights are used later to compute scores used in ranking/pruning of alarms. As opposed to these studies, our literature survey studies approaches that have been proposed for postprocessing of alarms.

As compared to the existing reviews and studies which aimed at understanding postprocessing of alarms, our presented study is based on more papers and describes multi-level categorization of the approaches. Therefore, it can help the users and designers/developers of ASATs to choose the postprocessing approaches suited in their work.

2.8 Conclusion and Future Work

In this chapter, we have studied approaches proposed in the literature to postprocess alarms generated by ASATs. We conducted systematic literature search, by combining keywords-based database search and snowballing, to collect research papers that propose techniques for postprocessing of alarms. Through the literature search, we identified 130 papers relevant to the topic.

We reviewed the papers, and extracted and categorized the approaches proposed by them. Our categorization shows that,

> Six main categories of the approaches, namely clustering, ranking, pruning, AFPE, combination of static and dynamic analyses, and simplification of manual inspection, have been proposed for postprocessing of alarms.

During the categorization, wherever appropriate, the main categories are further categorized into multiple sub-categories depending on the techniques used to implement those approaches.

We have summarized the merits and shortcomings of these categories (Section 2.5) to assist users and designers/developers of ASATs to make informed choices. The categorized postprocessing approaches, being complementary, provide an opportunity to combine them. We observe that,

> The identified approaches can be combined together in several ways, and feasibility of the different combinations, their advantages and disadvantages, however, need to be studied.

Several of the identified categories—sound clustering, ranking, AFPE, simplification of manual inspection—can help ASATs that are used for code proving. Considering our work in industry to verify safety-critical systems, we studied the techniques implementing those categories of the approaches to identify areas of improvement. Our observations from this study led to the work described in the next chapters.

Chapter 3

Repositioning of Alarms

The large number of alarms reported by static analysis tools is often recognized as one of the major obstacles to industrial adoption of such tools.

We present repositioning of alarms, a novel automatic postprocessing technique intended to reduce the number of reported alarms without affecting the errors uncovered by them. The reduction in the number of alarms is achieved by moving groups of similar alarms along the control flow to a program point where they can be replaced by a single newly created alarm, called repositioned alarm*. In the repositioning technique, as the locations of repositioned alarms are different from locations of the errors uncovered by them, we also maintain traceability links between the repositioned alarms and the corresponding original alarms. The presented technique is tool-agnostic and orthogonal to many other techniques available for postprocessing of alarms.*

To evaluate the technique, we applied it as a postprocessing step to alarms generated for four verification properties on 16 open source and four industry applications. The results indicate that the alarms repositioning technique reduces the number of alarms by up to 20% over state-of-the-art alarms clustering techniques with median reduction of 7.25%.

3.1 Introduction

Static analysis tools have shown promise in automated detection of code anomalies and programming errors [12, 14, 21, 206, 228]. However, these tools often generate a large number of alarms, i.e., warning messages notifying the tool-user about potential errors [20, 92, 139, 186]. A high percentage of these alarms are false positives, i.e., alarms that do not represent an error. To partition alarms into false positives and true errors, their postprocessing, often manual, is inevitable [54, 84]. The large number of alarms generated and the cost involved in partitioning them manually have been recognized as major concerns in adoption of static analysis tools [20, 37, 92, 120].

Clustering of alarms, one of the six categories of approaches that we identified in the literature study (Chapter 2), is commonly used to reduce number of alarms. State-of-the-art clustering

techniques [71, 124, 150, 223] reduce the number of alarms by creating groups of similar alarms[1] and identifying several alarms in each group as dominant alarms of the group. However, there are instances where the clustering techniques fail to group similar alarms, e.g., similar alarms which belong to different branches of an *if* statement. This limitation is further illustrated in Section 3.2. To address this limitation, we ask the following research question.

RQ 2: How can we automatically group similar alarms that state-of-the-art alarms clustering techniques fail to group?

To overcome the limitations of state-of-the-art clustering techniques, motivated by the work of Gehrke et al. [71], we propose to reposition alarms. Repositioning aims at reducing the number of alarms by moving groups of similar alarms along the control flow to a program point where they can be replaced by a single newly created alarm. We call the alarms generated by a static analysis tool *original alarms*, and the new alarms that we create *repositioned alarms*. We perform repositioning of alarms with the following primary goal.

To reduce the number of alarms without reducing the number of errors detected by them.

Furthermore, since traditional static analysis tools report alarms at the locations where run-time errors are likely to occur, the user has to traverse the code back to the causes of an alarm to identify whether the alarm represents an error [54, 101, 148]. Given the large size and complexity of industrial source code [148], this traversal can be a daunting task. To reduce these code traversals, we therefore, through repositioning, also aim:

To report alarms closer to their causes.

As the locations of repositioned alarms are different from locations of the errors detected by them, we also maintain traceability links between repositioned alarms and their corresponding original alarms. The traceability links are useful during manual inspection of the repositioned alarms, and get explored by the user only when a repositioned alarm is found to uncover an error.

We implement alarms repositioning by propagating *alarm conditions*—checks performed by the analysis tools—first in the backward direction and later in the forward direction. The propagation of conditions is through computation of *anticipable* and *available* conditions respectively using data flow analyses [100, 163, 193].

Our repositioning technique is tool-agnostic and orthogonal to many other techniques available for postprocessing of alarms. The technique is suitable for alarms reported by analysis tools that compute flow of values of the program variables, e.g., Polyspace Code Prover [1] and Frama-C [43]. We do not consider alarms reported based on structural information or local bug patterns [88].

We performed empirical evaluation of the proposed technique using 33,162 alarms generated by a commercial static analysis tool, TCS ECA [197], on 16 open source and four industry applications. The open source applications were selected from the benchmarks used to evaluate earlier alarms clustering techniques [124, 223]. The industry applications were embedded

[1] Two alarms are said to be *similar* if the checks performed by them are same (Section 3.2.2).

systems belonging to the automotive domain. Before performing repositioning, the input alarms were processed using state-of-the-art clustering techniques [124, 150, 223]. We observed that the proposed repositioning of alarms reduces the number of alarms by up to 20% over the existing clustering techniques, with median and average reduction being 7.25% and 6.47%, respectively.

The key contribution of this chapter is a novel and empirically evaluated postprocessing technique that reduces the number of alarms by repositioning. Repositioning reduces the number of alarms while detecting the same number of errors.

Chapter Outline Section 3.2 presents an informal overview of the repositioning technique using a motivating example. Section 3.3 describes repositioning of alarms formally, while Sections 3.4 and 3.5 describe two data flow analyses used to reposition alarms. Section 3.6 discusses our experimental evaluation. Section 3.7 presents related work, and Section 3.8 concludes.

3.2 Repositioning of Alarms

In this section, we informally describe repositioning as means to reduce the number of alarms. We first recapitulate the notions related to the control flow graph (CFG) of a program.

3.2.1 Background: Control Flow Graph

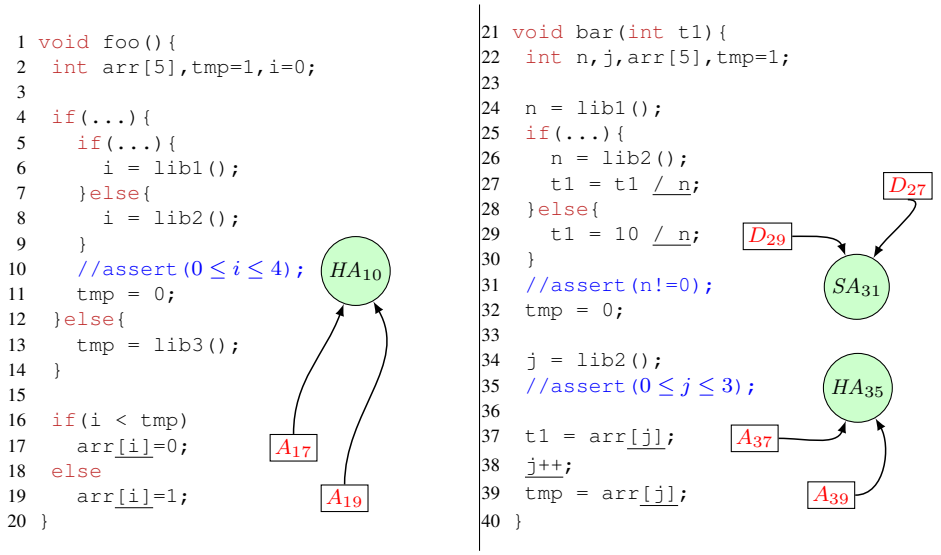
A control flow graph (CFG) [5] of a program is a directed graph $\langle \mathcal{N}, \mathcal{E} \rangle$, where $\mathcal{N}$ is a set of nodes representing the program statements (like assignments and controlling conditions); and $\mathcal{E}$ is a set of edges where an edge $\langle n, n' \rangle$ represents a possible flow of program control from $n \in \mathcal{N}$ to $n' \in \mathcal{N}$. A CFG has two distinguished nodes *Start* and *End*, representing the entry and exit of the corresponding program, respectively. 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, i.e., the logical formuala, corresponding to a conditional edge $u \rightarrow v$ as *cond*$(u \rightarrow v)$.

Except for the *Start* and *End* nodes, we assume that there is a one-to-one correspondence between the CFG nodes and their corresponding statements in the program. Thus, we use the terms statement and node interchangeably. Henceforth in code examples, we show only one statement per line and use n_m to denote the node of the program statement at line m. We assume that each of the program statements is reachable and does not cause side effects[2]. For a given node n, we use *pred*(n) (resp. *succ*(n)) to denote predecessors (resp. successors) of n in the graph.

Due to the one-to-one correspondence between the CFG nodes and their corresponding statements, the entry and exit of a node respectively correspond to the location just before and immediately after execution of the node's corresponding statement. We use *entry*(n) and *exit*(n) to respectively denote the *entry* and *exit* of a node n, i.e., the locations *just before* and *immediately after* execution of the statement corresponding to the node n. For every node n, *entry*(n) and *exit*(n) are referred to as *program points*.

A path from a node $n_i \in \mathcal{N}$ to node $n_k \in \mathcal{N}$ is a sequence of nodes n_i, n_{i+1},..., n_k such that for every consecutive pair of nodes (n_j, n_{j+1}) in the path there is an edge from n_j to n_{j+1}. For every node appearing on a path from a node $n_i \in \mathcal{N}$ to node $n_k \in \mathcal{N}$, we assume that

[2] A side effect is any change in program state that occurs as a by-product of the evaluation of an expression/statement [79]. For example, $x = y$++; is a statement with side effect, because it increments value of y in addition to evaluating the assignment expression (assigning y to x).



(1) *lib1*, *lib2*, and *lib3* are library functions whose code is not available for static analysis and their return-type is *signed int*.
(2) Ellipsis (...) indicates the code omitted for simplifying the example.

Figure 3.1: Alarm examples with their repositioning

the entry and exit of the node also appear on the path. Therefore, paths from a program point p to a program point q are same as the paths from the node corresponding to p to the node corresponding to q.

A node d *dominates* a node n if every path from the *Start* node to n also passes through d. On similar lines, a program point p_1 *dominates* a program point p_2 if every path from *entry*(*Start*) (i.e., 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 *exit*(*End*) (i.e., the *program exit*) contains p_1.

3.2.2 Motivating Example

Consider the *C* code example in Figure 3.1 adapted from a real-life embedded system. The code is simplified considerably but it is still sufficiently rich to illustrate repositioning of alarms. The example includes two functions, *foo* and *bar*, independent of each other. Analyzing the code for two commonly checked categories of runtime errors, array index out of bounds (AIOB) and division by zero (DZ), using a deep static analysis tool such as Polyspace Code Prover [1] or Frama-C [43] generates six alarms. We use notations A_n and D_n, respectively, to denote an alarm at line n corresponding to AIOB and DZ. The alarms are generated because values returned by the calls to library functions are treated as unknown by the analysis tool. The alarms are reported at the locations where run-time errors are likely to occur. We refer to these tool generated alarms as *original alarms* and their locations as *original locations*.

Definition 3.2.1 (Alarm Condition)**.** Given an alarm ϕ, assuming the program point of ϕ is reachable, the *alarm condition* of ϕ is a formula which is *true* if and only if ϕ is a false positive. ■

For example, the alarm condition of A_{17} (resp. D_{27}) shown in Figure 3.1 is $i \geq 0$ && $i \leq 4$ (resp. n != 0): this formula (condition) is true if and only if the alarm is a false positive. Note that the alarm condition of any alarm is a semantically unique formula. For an alarm ϕ, we use *cond*(ϕ) to denote its alarm condition.

Definition 3.2.2 (Similar Alarms)**.** Two alarms ϕ and ϕ' are called *similar* if and only if *cond*(ϕ) $\Leftrightarrow$ *cond*(ϕ'). ■

For example, alarms A_{17} and A_{19} in Figure 3.1 are similar.

Definition 3.2.3 (Cause Point of an Alarm)**.** An assignment statement is said to be a *cause point* of an alarm if the values assigned in this statement directly or indirectly reach to the variable(s) in the alarm condition, and these values are a reason for generating the alarm. ■

In other words, the statements in which the unknown (over-approximated) or unsafe values relevant to the static analysis originate and result in generation of an alarm are called cause points of the alarm [154]. For example, the assignment statements at lines 6 and 8, involving the calls to library functions, are cause points of A_{17} and A_{19}. The values returned by the calls to library functions are treated as unknown by the analysis tool, and these values ultimately result in generation of the two alarms.

Definition 3.2.4 (Dominant and Follower Alarms)**.** An alarm ϕ_1 is called a *dominant alarm* of an alarm ϕ_2 if, whenever ϕ_1 is a false positive, ϕ_2 is also a false positive. When an alarm ϕ_1 is *dominant alarm* of an alarm ϕ_2, ϕ_2 is called a *follower alarm* of ϕ_1. ■

State-of-the-art clustering techniques [124, 150, 223], which aim to reduce the number of alarms, group alarms based on their similarity or correlation. These techniques achieve the reduction by identifying dominant and follower alarms, and grouping each dominant alarm together with its follower alarms. The clustering techniques are, however, unable to group similar alarms A_{17} and A_{19} in Figure 3.1, because the alarms are reported in the two different branches of the *if* statement at line 16 and hence neither of them can be identified as a dominant alarm for the other. In this case, both A_{17} and A_{19} get reported as dominant alarms. Similarly, D_{27} and D_{29}, that are similar, do not get grouped together because they are reported in the two different branches of the *if* statement at line 25. Furthermore, these two alarms are caused by different reasons: the assignments at lines 26 and 24 respectively.

To overcome the above described limitation of alarms clustering techniques, Gehrke et al. [71] use propagation of alarm conditions. First, alarm conditions of original alarms are propagated backward along the control flow, and the propagated conditions are used to create new alarms at locations different from the original locations. Later, alarm conditions of the newly created alarms, propagated forward along the control flow, are used to remove the original alarms. However, as new alarms are created when no more upward propagation of the conditions is possible, the number of alarms reported finally can increase. For example, for A_{17} and A_{19} in Figure 3.1 the approach creates three new alarms: two at locations immediately after lines 6 and 8, and one immediately before line 13. Also, applying this approach to D_{27} and D_{29} does not help in reducing their number. Moreover, the locations of newly created alarms are different from the

locations of the errors they detect. Therefore, when a newly created alarm denotes an error, it requires user's effort to locate the actual error program point.

Our approach to reposition alarms with the two goals (Section 3.1) is motivated by the work of Gehrke et al. [71]. The approach overcomes the following limitations of state-of-the-art alarms clustering techniques [71, 124, 150, 223]: (1) group similar alarms that the techniques fail to group, (2) report alarms closer to their causes, and (3) report traceability links from the newly created (repositioned) alarms to their corresponding original alarms. We first describe the approach proposed using examples of alarms shown in Figure 3.1.

3.2.3 Hoisting of alarms

Consider the similar alarms A_{17} and A_{19}: both are AIOB alarms and based on the same variable i. These two alarms get generated because the values assigned to i at lines 6 and 8 are unknown. As the two alarms have the same alarm condition and the same cause points, they can be merged together. The new alarm after their merging can be created at line 10 where the paths coming from the two cause points meet for the first time. We call such a newly created alarm *repositioned alarm*, and denote it using *assertion* whose condition is same as the condition of the alarm. The repositioned alarm created for A_{17} and A_{19} is shown as an assertion HA_{10}, where $cond(HA_{10})$ is $i \geq 0$ && $i \leq 4$. During the repositioning of A_{17} and A_{19}, the effect of the *else* branch at line 12 is ignored, because (1) value of i is 0 if the *else* branch at line 12 is taken (due to the assignment at line 2) and the alarms A_{17} and A_{19} are safe due to this value; and (2) we are not interested in the scenarios in which the alarms are guaranteed to be safe.

We refer to repositioning of an alarm to a program point earlier in the code as *hoisting*, and an alarm resulting after the repositioning as a *hoisted alarm*. The hoisting of A_{17} and A_{19} at line 10 is safe, because HA_{10} is a false positive if and only if A_{17} and A_{19} are false positives. Thus, reporting HA_{10} instead of A_{17} and A_{19} is sufficient for error detection.

Note that hoisting of A_{17} and A_{19} is also possible and safe at line 15, however we prefer the hoisting at line 10, because the alarm reported at line 10 is closer to its cause points at lines 6 and 8: recall that we also aim to reduce backward code traversals performed during manual inspections of alarms. For example, inspecting a repositioned alarm at line 15 (or even A_{17} or A_{19}) requires traversing the code from line 15 backwards through the *then* branch to assignments in lines 6 and 8, and through the *else* branch at line 12 back to the assignment at line 2. Inspecting the hoisted alarm HA_{10} eliminates the need of inspecting the *else* branch. The gain achieved due to eliminating such code traversals, can be even bigger when the original and hoisted alarms belong to different functions.

Note that the possible hoisting of A_{17} and A_{19} closest to their cause points, is immediately after the assignments at lines 6 and 8. However, doing so results in two repositioned alarms and it does not allow us to reduce the number of alarms. Thus, we prefer hoisting A_{17} and A_{19} at line 10 and this hoisting is optimal considering the two alarms repositioning goals. Such alarms repositioning not only reduces the number of alarms but also reduces code traversals, to some extent, performed during the manual inspections.

We stress that, in the absence of either A_{17} or A_{19}, the other alarm cannot be safely hoisted to a location earlier in the code (e.g. to line 15), as the hoisting is not *outcome preserving*: the hoisted alarm can represent an error while the original alarm being a safe point.

We describe another example of alarms hoisting by referring to A_{37} and A_{39} that are similar. State-of-the-art clustering techniques fail to identify any one of them as a dominant alarm for the other due to the increment operation at line 38, resulting in reporting of both the alarms as dominant alarms. We observe that both the alarms can be safely merged into a single alarm

repositioned at line 35, denoted by HA_{35}. Note that *cond*(HA_{35}), i.e. $j \geq 0$ && $j \leq 3$, is such that HA_{35} is a false positive *if and only if* A_{37} and A_{39} are false positives. In this scenario, the repositioning allows to reduce the number of alarms by one. Furthermore, it also eliminates inspecting the second alarm, which requires considering the effect of the increment operation at line 38. The saving achieved in inspection effort can be considerable if the code at the three lines 37-39 appear in different functions.

3.2.4 Sinking of alarms

Hoisting of alarms achieves both the repositioning goals. However, it may not always help to merge alarms which can be merged together for reducing their number. For example, the original alarms D_{27} and D_{29} are candidates for repositioning as they are similar and also they appear in different branches of the *if* statement at line 25. They cannot be merged and hoisted before the *if* statement at line 25 as doing so misses capturing the effect of the cause point at line 26. In such cases, repositioning them later in the code at line 31 allows to merge them together while capturing the effect of both the cause points at lines 24 and 26. This repositioning helps to reduce the number of alarms by one. The alarm after the repositioning is shown as SA_{31}, and *cond*(SA_{31}) is $n \neq 0$. This repositioning is safe, because SA_{31} is a false positive *if and only if* D_{27} and D_{29} are false positives.

We refer to this type of repositioning down the control flow as *sinking of alarms*, and a repositioned alarm resulting after the sinking as *sunk alarm*. Since there can exist multiple program points for safe sinking of alarms (as it is the case at line 31 onwards for alarms D_{27} and D_{29}), we select the highest program point where paths coming from the alarms meet for the first time (i.e., at line 31 for D_{27} and D_{29}). The selection of the highest program point is to report sunk alarms closer to their corresponding original alarms. Note that although the sinking reduces the number of alarms by one (as per the primary goal), the sunk alarm SA_{31} is further away from the alarm cause points. Thus, we perform sinking of alarms only if it reduces more alarms than their hoisting.

3.2.5 Maintaining Traceability Links

In the repositioning technique, as the locations of repositioned alarms are different from locations of errors detected by the alarms, we maintain traceability links between a repositioned alarm and its corresponding original alarm(s). These links will be explored by users only when a repositioned alarm is found to uncover an error during manual inspection and a correction is needed at the program point(s) of its corresponding original alarms.

3.3 Technique Overview

This section provides overview of our alarms repositioning technique. We begin by defining a few terms that we use to describe the technique.

3.3.1 Definitions

Similarly to available and anticipable expressions [100, 163], we define *available alarm conditions* and *anticipable alarm conditions*. We use notation ϕ_p to denote an original alarm ϕ reported at a program point p.

Definition 3.3.1 (Available Alarm Conditions)**.** An alarm condition c is *available* at a program point p, if every path from the program entry to p contains an alarm ϕ_q with c as its alarm condition, and the point q is not followed by an assignment to any variable in c on any path from q to p. ■

For example, in Figure 3.1, condition $n \neq 0$ is an available alarm condition at all the program points after *entry*(n_{32}) due to the alarms D_{27} and D_{29}.

Definition 3.3.2 (Anticipable Alarm Conditions)**.** An alarm condition c is *anticipable* at a program point p, if every path from p to the program exit contains an alarm ϕ_q with c as its alarm condition, and the point q is not preceded by an assignment to any variable in c on any path from p to q. ■

For example, in Figure 3.1, condition $i \geq 0$ && $i \leq 4$ is an anticipable alarm condition at *entry*(n_{13}), *exit*(n_6), and *exit*(n_8) due to the alarms A_{17} and A_{19}.

Definition 3.3.3 (Safe Repositioning of Alarms)**.** A repositioning (hoisting or sinking) of a set of original alarms S, resulting in a set of repositioned alarms, is said to be *safe* if the following holds: each repositioned alarm is a false positive if and only if its corresponding original alarms in S are false positives. ■

3.3.2 Repositioning Technique

To perform repositioning of alarms, discussed in the previous section, we design a two step static analysis technique as described next. In our technique, we accept all the tool-generated (original) alarms as input, and do not explicitly identify groups of similar alarms prior to their repositioning.

3.3.2.1 Intermediate Repositioning

In the first step, alarm condition of every original alarm ϕ reported at p is safely hoisted at the *highest hoisting point along every path* that reaches p. The highest hoisting point on a path is identified as the program point q_h such that *cond*(ϕ) is anticipable at q_h but the same condition is no longer anticipable at any program point just before q_h. Thus, this step results in hoisting an alarm condition closer to its cause points, but it can also result in hoisting the same condition at multiple locations. For example, the alarm condition $i \geq 0$ && $i \leq 4$ of alarms A_{17} and A_{19} in Figure 3.1 gets hoisted at two locations: *exit*(n_6) and *exit*(n_8). This step discards the third hoisting possible at *entry*(n_{13}), because the hoisted condition (repositioned alarm) is a false positive: the value of i at this point is always 0 (due to the assignment at line 2) and the hoisted condition never evaluates to *false*. Note that the repositioning obtained after this step is not final and it requires improvement. Thus, we refer to it as *intermediate repositioning*. Section 3.5 describes this first step in detail.

3.3.2.2 Improvement of the Intermediate Repositioning

This step improves the intermediate repositioning by merging hoisted alarm conditions that are candidates for sinking. In this step, *available alarm conditions* are computed from alarm conditions hoisted in the intermediate repositioning. For every available alarm condition c computed at a program point p, we also compute exactly one program point p_d to be associated with c. The associated point p_d is the highest program point among the program points where c is available

and their nodes dominate the node of p. The point p_d is used to reposition c during the final repositioning.

To compute the associated point p_d for an available condition c, initially c is associated with the single program point at which it gets generated: a hoisting location from the intermediate repositioning. Later, at a *meet-program point* p_m where c is observed to have two or more different program points associated with it, the association of c is updated to p_m. With this operation we guarantee that, for every condition c available at a program point p, there exists only one program point associated with c and the node of the associated point dominates the node of p.

For example in Figure 3.1, $c := i \geq 0$ && $i \leq 4$ gets generated as available alarm condition at the hoisting locations *exit*(n_6) and *exit*(n_8) in the intermediate repositioning example (refer Section 3.3.2.1). At its generation point *exit*(n_6) (resp. *exit*(n_8)), c gets associated with the same program point. As *entry*(n_{11}) is a meet point where c is available and also has those two program points *exit*(n_6) and *exit*(n_8) associated with it, the association of c is changed to *entry*(n_{11}). The location *entry*(n_{11}) associated with c is used later to reposition the condition c finally, described in Section 3.5.

3.4 Intermediate Repositioning

In this section, we describe performing the first step of the repositioning technique, i.e., the *intermediate repositioning of alarms* (Section 3.3.2.1). We first describe computation of anticipable alarm conditions from the alarms input for repositioning. The computation of anticipable alarm conditions is using a backward data flow analysis. Later, we describe obtaining the intermediate repositioning—temporary hoisting of alarms—using results of the backward analysis, i.e., the anticipable alarms conditions computed.

3.4.1 Anticipable Alarm Conditions Analysis

Given CFG of a program and original alarms set Φ, this backward data flow analysis computes *anticipable alarm conditions* (antconds). We call this analysis *antconds analysis*. For every antcond c computed at any program point, antconds analysis also computes the input alarms, $\Phi' \subseteq \Phi$, which contribute to anticipability of c at that point. We refer to these alarms Φ' as *related original alarms* (rel-alarms) of antcond c. The rel-alarms are used to compute traceability links between a *repositioned condition* and its corresponding original alarm(s). Henceforth, in this chapter, we use repositioned condition to refer to an alarm repositioned in the intermediate or final repositioning, to distinguish it from the original alarms.

3.4.1.1 Notations

Let P be the set of all program points, i.e., *entry*(n) and *exit*(n) of every node n in the CFG of the program are elements of P. Let V be the set of variables in the program, and C be the set of all conditions that can be formed using program variables, constants, and arithmetic and logical operators. We use tuple $\langle c, \phi \rangle$ to denote an antcond $c \in$ C along with one of its rel-alarms $\phi \in \Phi$. Thus, the values computed by antconds analysis at a program point are given by a subset of $\mathrm{L}_b = \mathrm{C} \times \Phi$. For a given set $S \subseteq \mathrm{L}_b$, we define the following functions.

- *condsIn*$(S) = \{c \mid \langle c, \phi \rangle \in S\}$, returns all antconds in S; and

- *tuplesOf*$(c, S) = \{\langle c, \phi\rangle \mid \langle c, \phi\rangle \in S\}$, returns all tuples of a given antcond $c \in S$.
- *alarmsOf*$(c, S) = \{\phi \mid \langle c, \phi\rangle \in S\}$, returns all rel-alarms of $c \in S$, i.e., all original alarms associated with c.

For a given $c \in$ C, and $C' \subseteq$ C, as shown below we define function *getImplyingCond*(c, C') to return a *unique condition* that implies c.

$$\textit{getImplyingCond}(c, C') = \begin{cases} \textit{getUniqueCond}(c, C') & c' \in C',\ c' \Rightarrow c \\ c & \text{otherwise} \end{cases}$$

$$\textit{getUniqueCond}(c, C') = \textit{getFirstLexicOrdered}(\{\ c_1 \mid c_1 \in C',\ c_1 \Rightarrow c,\ \nexists c_2 \in C',\ c_2 \Rightarrow c_1\})$$

where the function *getFirstLexicOrdered*(X) is assumed to return the first element when elements in set X are ordered lexicographically.

3.4.1.2 Lattice

Antconds analysis aims at computing subsets of L_b flow-sensitively at every program point $p \in$ P. Let B' be the powerset of L_b. To define the lattice of antconds analysis, we introduce an artificial top element $\top$. We assume that the element has the following properties: *condsIn*$(\top) =$ C; and $\forall c \in$ C: *alarmsOf*$(c, \top) = \emptyset$.

Let $B = B' \cup \{\top\}$. The lattice of the values computed by antconds analysis is given by $\langle B, \sqsubseteq, \top, \bot, \sqcap\rangle$, where (a) the *top* and *bottom* of the lattice respectively are $\top$ and $\bot = \emptyset$; and (b) the *partial order* $\sqsubseteq$ and the *meet* operation $\sqcap$ are as described below.

Given $X, Y \in B, X \sqsubseteq Y$ iff

$$\textit{condsIn}(X) \subseteq \textit{condsIn}(Y) \wedge (\forall c \in \textit{condsIn}(X):\ \textit{alarmsOf}(c, X) \supseteq \textit{alarmsOf}(c, Y)).$$

In other words, when $Y \neq \top$, $X \sqsubseteq Y$ iff $\forall c \in$ *condsIn*(X): $\langle c, \phi\rangle \in X \Rightarrow \langle c, \phi'\rangle \in Y \wedge$ *alarms*$(c, X) \supseteq$ *alarmsOf*(c, Y). When $Y = \top$, $X \sqsubseteq Y$ holds by the definition of $\top$. As the partial order is based on union and intersection operations, it is easy to see that the partial order $\sqsubseteq$ is reflexive, antisymmetric, and transitive. We use $\sqcap_B$ to denote the meet operation $\sqcap$, and is defined by Equation 3.1.

Given $X, Y \in B$,

$$X \sqcap_B Y = \bigcup_{c \in \left(\textit{condsIn}(X)\ \cap \textit{condsIn}(Y)\right)} \textit{tuplesOf}(c, X)\ \cup\ \textit{tuplesOf}(c, Y) \tag{3.1}$$

The meet operation (Equation 3.1) first computes intersection of antconds in X and Y, and then for each of the antconds in the result, it performs union of its rel-alarms (i.e., union of its tuples). This meet operation is commutative, associative, and idempotent. The meet $\sqcap_B$ and join $\sqcap$ operations are dual to one another with respect to order inversion.

For the lattice defined above, the following holds:
$\forall X \in B$: $X \sqcap \top = X,\ X \sqcup \top = \top,\ X \sqcap \bot = \bot,\ X \sqcup \bot = X$.

As the meet $\sqcap_B$ is commutative, associative, and idempotent; the partial order $\sqsubseteq$ is reflexive, antisymmetric, and transitive; and there exists the bottom element $\bot$, all strictly descending chains are finite [100].

Let $m, n \in \mathrm{N}$; $c, c' \in \mathrm{C}$; $\phi, \phi' \in \Phi$; $X \in B$; and $Y \subseteq \mathrm{C}$.

$$\mathit{AntOut}_n = \begin{cases} \emptyset & n \text{ is } \mathit{End} \text{ node} \\ \bigsqcap_{m \in succ(n)}^{B} \mathit{AntIn}_m & \text{otherwise} \end{cases} \tag{3.2}$$

$$\mathit{AntIn}_n = \mathit{Gen}_n(\mathit{AntOut}_n) \cup (\mathit{AntOut}_n \setminus \mathit{Kill}_n(\mathit{AntOut}_n)) \tag{3.3}$$

$$\mathit{Kill}_n(X) = \{ \langle c, \phi \rangle \mid \langle c, \phi \rangle \in X,\ n \text{ contains a definition of an operand of } c \} \tag{3.4}$$

$$\mathit{Gen}_n(X) = \mathit{Gen'}_n(X) \cup \mathit{DepGen}_n(\mathit{Kill}_n(X)) \tag{3.5}$$

$$\mathit{Gen'}_n(X) = \{ \mathit{process}(\phi, \mathit{condsIn}(X)) \mid \text{alarm } \phi \in \Phi \text{ has been reported for } n \} \tag{3.6}$$

$$\mathit{process}(\phi, Y) = \langle \mathit{getImplyingCond}(\mathit{cond}(\phi), Y), \phi \rangle \tag{3.7}$$

$$\mathit{DepGen}_n(X) = \begin{cases} \left\{ \langle \mathit{wprecond}(n, c), \phi \rangle \;\middle|\; \begin{array}{c} \langle c, \phi \rangle \in X, \\ \mathit{wprecond}(n, c) \text{ is computable} \end{array} \right\} & n \text{ is an assignment node} \\ \emptyset & \text{otherwise} \end{cases} \tag{3.8}$$

Figure 3.2: Data flow equations of the antconds analysis.

3.4.1.3 Data Flow Equations

The required *initialization* of the data flow values at every program point is with $\top$. Figure 3.2 shows data flow equations of antconds analysis in intraprocedural setting: handling of the call nodes is not shown for simplicity of the formalization. We use AntIn_n and AntOut_n, in Equations 3.3 and 3.2, to denote antconds computed by the analysis at the entry and exit of a node n respectively. Equation 3.2 indicates that, $\emptyset$ is the value computed at $\mathit{exit}(End)$ (boundary information), and values at the exit of every other node n, i.e., AntOut_n, are computed by performing meet of the values computed at the entry of every successor of n (shown using $\sqcap_B$). The solution computed by these equations is the maximum fixed point solution.

Equation 3.6 shows processing of every alarm ϕ reported for the statement of a node n, to generate $\mathit{cond}(\phi)$ as an anticipable condition with ϕ as its rel-alarm. Equation 3.7 denotes that the alarm condition $\mathit{cond}(\phi)$ is generated as an antcond only if no antcond in $\mathit{condsIn}(\mathit{AntOut}_n)$—the antconds flowing in at the exit of node n—implies $\mathit{cond}(\phi)$. In the other case, i.e., when $\mathit{cond}(\phi)$ is *implied by* an antcond c_{in} flowing in at the exit of node n, $\mathit{cond}(\phi)$ is not generated as an antcond. However in this case, ϕ is associated with c_{in} as its rel-alarm. This processing allows to compute an alarm ϕ at node n as a follower of the alarms due to which the implying condition c_{in} is an antcond at the exit of n.

Equation 3.8 denotes computation of antconds transitively, i.e., antconds at a node are also generated based on antconds that flow-in at the exit of the node. This equation assumes that the function $\mathit{wprecond}(n, c_p)$ returns the weakest precondition for (1) the statement of a given node n, and (2) a postcondition c_p. Note that, an antcond is transitively generated from an antcond c that gets killed at an assignment node n. Therefore, we compute weakest preconditions only for assignment nodes, and transitively generate antcond from a killed antcond c only if weakest precondition can be computed for c. We compute antconds transitively to group more alarms together: if antconds are not computed transitively, some of the alarms cannot be grouped through repositioning. For example, A_{37} and A_{39} in Figure 3.1 cannot be grouped together through their

repositioning at *exit*(n_{34}) (Section 3.2.3), if antconds are not computed transitively (at node n_{38}). In this case, the repositioning of A_{37} (resp. A_{39}) would be at *exit*(n_{34}) (resp. *exit*(n_{38})).

Equations 3.5 and 3.4 respectively compute antconds to be generated and killed at a node, based on antconds that flow in at the exit of the node. Hence, these equations indicate that the computation of antconds by these equations is non-constant. Moreover, since antconds are computed by taking into account implication and transitivity of the conditions, the flow function (framework) to compute antconds (Equation 3.3) is non-separable[3].

The flow functions (data flow equations) shown in Figure 3.2 are monotonic[4]. Monotonicity of the flow function that computes values at the exit of nodes (Equation 3.2), follows from the definition of the meet operation $\sqcap_B$. Therefore, below we describe monotonicity of the flow function that computes values at the entry of the nodes (shown by Equation 3.3). To prove this, we need to prove, $\forall X, Y \in B : X \sqsubseteq Y \Rightarrow (\mathit{Gen}_n(X) \cup (X \setminus \mathit{Kill}_n(X))) \sqsubseteq (\mathit{Gen}_n(Y) \cup (Y \setminus \mathit{Kill}_n(Y)))$. Consider an antcond $c \in \mathit{condsIn}(X)$. Therefore, $c \in \mathit{condsIn}(Y)$ (because $X \sqsubseteq Y$). If c gets killed at a node n, it will be removed along with its rel-alarms from both X and Y. Moreover, if c implies the condition of an alarm at node n, the same alarm will get added as a rel-alarm of c in both X and Y. On similar lines, for the other possibilities with c—transitivity and no effect of node n on c—there will be same effect on both X and Y by the flow function. Moreover, when a new antcond is generated from an alarm reported at n (when no flowing-in antcond implies the alarm's condition), the same antcond gets added, to X and Y, with the same alarm as its rel-alarm. Therefore, for every $c \in \mathit{condsIn}(Y)$, the flow function has same effect on both X and Y, indicating $(\mathit{Gen}_n(X) \cup (X \setminus \mathit{Kill}_n(X))) \sqsubseteq (\mathit{Gen}_n(Y) \cup (Y \setminus \mathit{Kill}_n(Y)))$.

3.4.2 Intermediate Repositioning of Alarms

Recall the intermediate repositioning (Section 3.3.2.1) is implemented by hoisting alarm condition of every alarm *temporarily* at the highest program point in every path reaching to the alarm program point. The highest program point for hoisting is identified as the point before which the alarm condition is no longer anticipable. We distinguish between the two possible cases of identifying the highest hoisting points.

3.4.2.1 Case 1 (Hoisting at Entry of Nodes)

An alarm condition c anticipable at *entry*(n) is not anticipable at *exit*(m), where m is a predecessor of n and also a branching node (i.e. the statement of m is a controlling condition). This case occurs when a branch coming out of m exists where c is not anticipable at the start of the branch. In this case, the antcond c is hoisted at *entry*(n). Alternatively, the antconds to be hoisted at *entry*(n) are given by

$$\mathit{Hoist'}_{\mathit{entry}(n)} = \mathit{condsIn}\,(\mathit{AntIn}_n) \;\setminus \bigcap_{m \in pred(n)} \mathit{condsIn}\,(\mathit{AntOut}_m) \tag{3.9}$$

For example, condition $i \geq 0$ && $i \leq 4$ of A_{17} (and A_{19}) in Figure 3.1 is anticipable at *entry*(n_{13}) but not at *exit*(n_4), and n_4 is predecessor of n_{13}. Thus, the condition is hoisted at *entry*(n_{13}).

[3] A flow function $f : L \mapsto L$ is *separable* iff it is a tuple $< \langle \widehat{f}^{\alpha}, \widehat{f}^{\beta}, ..., \widehat{f}^{\omega} \rangle$ of component functions $\widehat{f} : \widehat{L} \mapsto \widehat{L}$. If $\widehat{f}$ is of the form $L \mapsto \widehat{L}$, then f is non-separable [100].

[4] Flow functions F are *monotonic* if $\forall f \in F$, $\forall X, Y \in B : X \sqsubseteq Y \Rightarrow f(X) \sqsubseteq f(Y)$.

3.4.2.2 Case 2 (Hoisting at Exit of Nodes)

A condition c is anticipable at *exit*(n) but not at *entry*(n) when (1) the node n contains a definition of an operand of c i.e. anticipability of c is killed by n, and (2) the node n does not generate any antcond transitively from c. In this case, we hoist the condition c at *exit*(n). That is, the alarm conditions to be hoisted at *exit*(n) are given by

$$\textit{Hoist'}_{\textit{exit}(n)} = \{c \mid \langle c, \phi\rangle \in \textit{Kill}_n\,(\textit{AntOut}_n)\;,\; \textit{DepGen}_n(\{\langle c, \phi\rangle\}) = \emptyset\} \tag{3.10}$$

As an example of the above hoisting case, the alarm condition of A_{17} (and A_{19}) in Figure 3.1, $i \geq 0$ && $i \leq 4$, is anticipable at *exit*(n_6) but not at *entry*(n_6) and n_6 does not generate any anticipable alarm condition transitively. Thus, the condition gets hoisted at *exit*(n_6). Similarly, the same condition also gets hoisted at *exit*(n_8).

3.4.2.3 Discarding Redundant Hoisting

Consider the hoisting of alarm condition of A_{17} and A_{19}, i.e., $i \geq 0$ && $i \leq 4$, at *entry*(n_{13}) (Section 3.4.2.1). The hoisted alarm condition always holds at the hoisting location due to the value 0 assigned to i at line 2. We deem this hoisting to be redundant and discard it. Formally, equations 3.11 and 3.12 compute the *non-redundant hoisting* of alarm conditions at *entry*(n) and *exit*(n), where *alwaysTrue*(c, p) is *true* only if the condition c always holds at p.

$$\textit{Hoist}_{\textit{entry}(n)} = \left\{c \;\middle|\; c \in \textit{Hoist'}_{\textit{entry}(n)},\; \textit{alwaysTrue}(c, \textit{entry}(n)) \neq true\right\} \tag{3.11}$$

$$\textit{Hoist}_{\textit{exit}(n)} = \left\{c \;\middle|\; c \in \textit{Hoist'}_{\textit{exit}(n)},\; \textit{alwaysTrue}(c, \textit{exit}(n)) \neq true\right\} \tag{3.12}$$

3.4.2.4 Algorithm

The intermediate repositioning of alarms—hoisting at the *entry* and *exit* of all the nodes—is performed by processing every node $n \in \mathcal{N}$ using the equations 3.11 and 3.12.

3.4.2.5 Computing Traceability Links

Recall that function *alarmsOf*(c, S) returns rel-alarms of c in $S \in B$ (Section 3.4.1.1). The traceability links are generated from a hoisted condition c in $\textit{Hoist}_{\textit{entry}(n)}$ (resp. $\textit{Hoist}_{\textit{exit}(n)}$) to its corresponding rel-alarms given by *alarmsOf*$(c, \textit{AntIn}_n)$ (resp. *alarmsOf*$(c, \textit{AntOut}_n)$).

3.4.2.6 Intermediate Repositioning Example

Following is the intermediate repositioning obtained for alarms in Figure 3.1.

1. $i \geq 0$ && $i \leq 4$ is hoisted at *exit*(n_6) and *exit*(n_8), with both A_{17} and A_{19} as its rel-alarms.

2. $n \neq 0$ is hoisted at *exit*(n_{26}) (resp. *entry*(n_{29})) with D_{27} (resp. D_{29}) as its rel-alarm.

3. $j \geq 0$ && $j \leq 4$ is hoisted at *exit*(n_{34}) with A_{37} as its rel-alarm.

4. Antcond $j \geq -1$ && $j \leq 3$, that gets transitively generated from antcond $j \geq 0$ && $j \leq 4$ that flows in at *exit*(n_{38}), is hoisted at *exit*(n_{34}) with A_{39} as its rel-alarm. The antcond $j \geq 0$ && $j \leq 4$ that is killed at line 38 is not hoisted at *exit*(n_{38}), because an antcond c killed at node n is not hoisted at *exit*(n) if a new antcond is transitively generated from c at the same node (Equation 3.10).

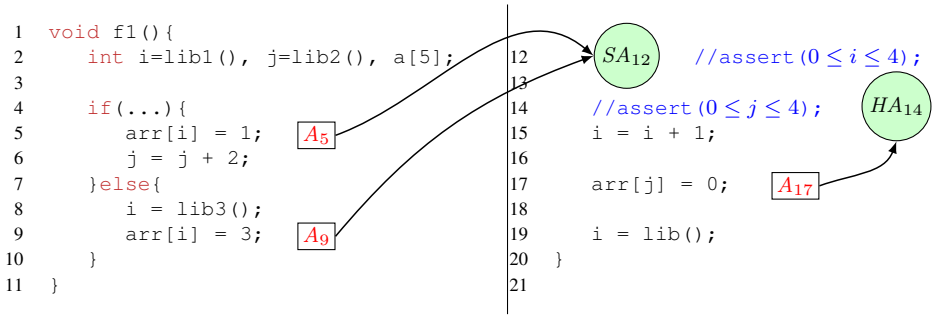


Figure 3.3: Examples to illustrate the requirement of (a) consistency in computation of antconds and avconds, and (b) computing repositioning conditions for avconds.

The two conditions hoisted in each of the cases (1) and (2) belong to different branches of an *if* statement and are candidates for merging (sinking) during the improvement step (described in the next section). The two conditions hoisted in the cases (3) and (4) are at the same program point ($exit(n_{34})$), and these two conditions get merged into a single condition during the improvement step.

3.5 Improvement of the Intermediate Repositioning

In this section, we describe performing the second step of the repositioning technique, i.e., *improvement of the intermediate repositioning*. We first describe computation of available alarm conditions from the conditions (alarms) hoisted in the intermediate repositioning. The computation of available alarm conditions is done using a forward data flow analysis. Similar to antconds analysis (Section 3.4.1), for each of the available alarm conditions (called *avconds*), we compute original alarms (rel-alarms[5]) that are used later to create traceability links from final repositioned alarms to their corresponding original alarms. Then, we describe the steps to create repositioned alarms using the results of this forward analysis and to postprocess the repositioned alarms.

3.5.1 Consistency in Antconds and Forward Analyses

Recall that the antconds are computed transitively (Section 3.4.1.3). The purpose of the transitive computation of antconds is to merge/group alarms that are always on the same paths and variable(s) in their conditions are modified between their program points. For example, recall the repositioning of the two similar alarms A_{37} and A_{39} at $exit(n_{34})$, shown in Figure 3.1 and discussed in Section 3.2.3. If antconds are not computed transitively, the two similar alarms A_{37} and A_{39} in Figure 3.1 cannot be merged and repositioned at $exit(n_{34})$, and their number cannot be reduced from two to one. Hence, the antconds are computed transitively (in the backward direction). If antconds are computed transitively, *avconds also need to be computed transitively* (in the forward direction): the computation of antconds and avconds need to be consistent. We illustrate this requirement—consistency in computation of antconds and avconds transitively—by referring to repositioning of A_{17} shown in Figure 3.3. The conditions hoisted in the intermediate repositioning, corresponding to A_{17} are: (a) $-2 \leq j \leq 2$ hoisted at $entry(n_5)$, and (b)

[5]In Section 3.4.1, we used rel-alarms to refer to related original alarms of an antcond. We use the same term to refer to the original alarms computed for an avcond.

$0 \leq j \leq 4$ hoisted at *entry*(n_8). During forward avconds analysis, if avconds are not computed transitively, the condition $0 \leq j \leq 4$ will not get computed as an avcond at *entry*(n_{15}) and even at *entry*(n_{17}). As a result, the two hoisted conditions (from the intermediate repositioning) will not get merged together, and therefore, each of them will be a separate repositioned alarm in the final repositioning (steps to obtain final repositioning are discussed in Section 3.5.4). If avconds are computed transitively, $0 \leq j \leq 4$ will be an avcond at *entry*(n_{15}) and later program points. Later, this avcond gets used to reposition $0 \leq j \leq 4$ at *entry*(n_{15}), i.e., A_{17} gets repositioned at *entry*(n_{15}) (shown as HA_{14}). This indicates that a single alarm can get repositioned at more than one location if antconds required to obtain intermediate repositioning are computed transitively, but avconds are not.

3.5.2 Computation of Avconds with Their Corresponding Rel-alarms

In order to compute traceability links from final repositioned alarms to their corresponding original alarms, for each of the avconds computed at a program point, its corresponding rel-alarms need to be computed. As this forward analysis computes avconds with their corresponding rel-alarms, we call it *rel-alarms analysis*. First we describe a data flow formalization of rel-alarms analysis: computing avconds with their corresponding rel-alarms. Then, we describe a method to create repositioned alarms using the analysis results. We stress that avconds and their corresponding rel-alarms are computed based on the conditions hoisted and their locations in the intermediate repositioning, rather than the original alarms input for repositioning and their locations.

3.5.2.1 Formalization of Rel-alarms Analysis

We present formalization of the forward analysis (*rel-alarms analysis*) using the notations used to describe antconds analysis (Section 3.4.1) and performing intermediate repositioning (Section 3.4.2). The lattice of the values computed by rel-alarms analysis is the same as the lattice of values computed by antconds analysis (Section 3.4.1.2). Hence, rel-alarms analysis computes subsets of B flow-sensitively at every program point and has the meet operation similar to Equation 3.1.

Figure 3.4 presents data flow equations of rel-alarms analysis that computes avconds transitively in an intraprocedural setting, along with their corresponding rel-alarms. The required *initialization* of the data flow values at every program point is with $\top$. Therefore, the solution computed by this analysis is the maximum fixed point solution. We use fwdIn_n and fwdOut_n, to denote avconds computed with their corresponding rel-alarms, respectively, at the entry and exit of every node n (Equations 3.13 and 3.14). Figure 3.5 illustrates the computation of fwdOut_n.

Equations 3.15 and 3.17 indicate that an avcond is generated from every condition hoisted in the intermediate repositioning. Similar to Equation 3.7 (Section 3.4.1.3), Equation 3.15 does not generate an avcond for $c \in \mathit{Hoist}_{\mathit{entry}(n)}$ when an avcond in $\mathit{condsIn}(\mathit{fwdIn}_n)$ implies c. However, such implication handling is not needed in Equation 3.17, because the antconds are hoisted at the exit of a node only when the node stops anticipability of those conditions. Equation 3.18 computes avconds transitively, where it assumes function $\mathit{postcond}(n, c_p)$ to return the *strongest postcondition* for the statement of a given node n, and a given precondition c_p.

On similar lines to Equations 3.5 and 3.4 of antconds analysis, Equations 3.15 and 3.16 respectively compute avconds to be generated and killed at a node, based on avconds that flow in at the entry of the node. Hence, these equations indicate that the computation of avconds by these equations is non-constant. Moreover, since avconds are computed by taking into account

Let $m, n \in \mathrm{N}$; $c, c' \in \mathrm{C}$; $\phi, \phi' \in \Phi$; $X \in B$; and $Y \subseteq \mathrm{C}$.

$$\mathit{fwdIn}_n = \begin{cases} \emptyset & n \text{ is } \mathit{Start} \text{ node} \\ \bigsqcap_{\substack{B \\ m \in pred(n)}} \mathit{fwdOut}_m & \text{otherwise} \end{cases} \tag{3.13}$$

$$\mathit{fwdOut}_n = \mathit{fwdOut'}_n \cup \mathit{Gen}_{\mathit{exit}(n)} \tag{3.14}$$

$$\mathit{fwdOut'}_n = (\mathit{fwdIn'}_n \setminus \mathit{Kill}_n(\mathit{fwdIn'}_n)) \cup \mathit{DepGen}(\mathit{fwdIn'}_n)$$

$$\mathit{fwdIn'}_n = \mathit{fwdIn}_n \cup \mathit{Gen}_{\mathit{entry}(n)}(\mathit{fwdIn}_n)$$

$$\mathit{Gen}_{\mathit{entry}(n)}(X) = \left\{ \mathit{createTuple}(c, \mathit{condsIn}(X), \phi) \;\middle|\; \begin{array}{c} c \in \mathit{Hoist}_{\mathit{entry}(n)}, \\ \phi \in \mathit{alarmsOf}(c, \mathit{AntIn}_n) \end{array} \right\} \tag{3.15}$$

$$\mathit{createTuple}(c, Y, \phi) = \langle \mathit{getImplyingCond}(\mathit{cond}(\phi), Y), \phi \rangle$$

$$\mathit{Kill}_n(X) = \{\, \langle c, \phi \rangle \mid \langle c, \phi \rangle \in X,\ n \text{ contains a definition of an operand of } c \,\} \tag{3.16}$$

$$\mathit{Gen}_{\mathit{exit}(n)} = \{\, \langle c, \phi \rangle \mid c \in \mathit{Hoist}_{\mathit{exit}(n)},\ \ \phi \in \mathit{alarmsOf}(c, \mathit{AntOut}_n) \,\} \tag{3.17}$$

$$\mathit{DepGen}_n(X) = \begin{cases} \left\{ \langle \mathit{postcond}(n, c), \phi \rangle \;\middle|\; \begin{array}{c} \langle c, \phi \rangle \in X, \\ \mathit{postcond}(n, c) \text{ is computable} \end{array} \right\} & n \text{ is an assignment node} \\ \emptyset & \text{otherwise} \end{cases} \tag{3.18}$$

Figure 3.4: Computing avconds with their corresponding related original alarms.

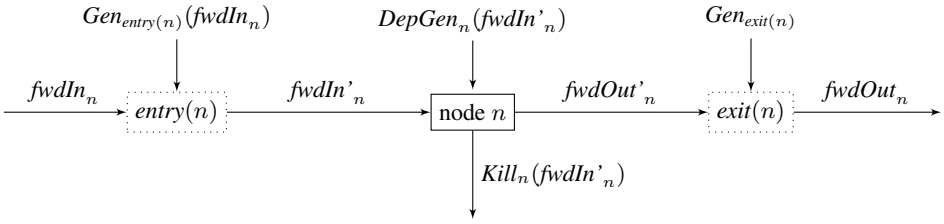


Figure 3.5: Processing of a node during (forward) rel-alarms analysis

implication and transitivity of the conditions, the flow function (framework) to compute avconds (Equation 3.14) is non-separable. Similar to the flow functions of antconds analysis in Figure 3.2, the flow functions (data flow equations) shown in Figure 3.4 are monotonic.

3.5.2.2 Computation of Repositioned Alarms

To create repositioned alarms using results of rel-alarms analysis, we use a method similar to the one we used to create repositioned alarms in the intermediate repositioning based on antconds. This method to create repositioned alarms based on avconds, identifies the conditions that are available at a program point but not immediately after it. That is, we identify avconds and the program points where their availability stops, and use them to create repositioned alarms. In addition to the avconds identified this way, as a special case, we also consider the avconds that reach to $\mathit{exit}(\mathit{End})$ without getting killed at any program point. This method is discussed later in Section 3.5.4.1.

Let c be a condition available at a program point p but not immediately after it. Recall that the intermediate repositioning is obtained by repositioning antconds at the locations before which their anticipability is stopped (Section 3.4.2). However, in this improvement step, for a condition whose availability stops after a program point p, we reposition c at the highest program point p_r, where (1) p_r dominates p, and (2) the same condition c is available at p_r and reaches from p_r to p. For example, consider alarm A_{17} shown in Figure 3.3. The conditions hoisted in the intermediate repositioning, corresponding to it (A_{17}), are: (a) $-2 \leq j \leq 2$ hoisted at *entry*(n_5), and (b) $0 \leq j \leq 4$ hoisted at *entry*(n_8). During rel-alarms analysis, due to transitive computation of avconds, condition $0 \leq j \leq 4$ reaches to *exit*(*End*) as an avcond. For this avcond reaching at *exit*(*End*), we identify the program point p_r where the same condition is available and p_r dominates *exit*(*End*). In this case, p_r = *entry*(n_{15}). Therefore, we reposition $0 \leq j \leq 4$ at *entry*(n_{15}); the resulting repositioned alarm is shown as HA_{14}.

Since we compute avconds transitively, an avcond c computed at a program point p can be a transformed version of a condition c_r available at the highest program point p_r (where we intend to reposition c). Thus, computing the highest program point p_r requires taking into account transitive computation of avconds. We illustrate this by referring to repositioning of alarms A_5 and A_9 shown in Figure 3.3. The conditions hoisted in the intermediate repositioning, corresponding to these two alarms, respectively are (a) $0 \leq i \leq 4$ hoisted at *entry*(n_5), and (b) $0 \leq i \leq 4$ hoisted at *exit*(n_8). When avconds are computed transitively, due to these two hoisted conditions in the intermediate repositioning, $0 \leq i \leq 4$ gets computed as an avcond *entry*(n_{15}), and $1 \leq i \leq 5$ as an avcond at *exit*(n_{15}), each of them having A_5 and A_9 as their rel-alarms. The avcond $1 \leq i \leq 5$, transitively generated at *exit*(n_{15}), gets propagated to and killed at *entry*(n_{19}). For the avcond $1 \leq i \leq 5$ killed at *entry*(n_{19}), if the transitivity is not taken into account, the above method identifies *exit*(n_{15}) as the location for its repositioning, i.e., p_r = *exit*(n_{15}). However, the intended repositioning is at *entry*(n_{15}). Note that avcond $1 \leq i \leq 5$ at *entry*(n_{19}) is a transformed version of avcond $0 \leq i \leq 4$ at *entry*(n_{15}). Therefore, in order to create a repositioned alarm at *entry*(n_{15}) (shown as SA_{12}), the corresponding avcond at that point needs to be identified.

Hence, to obtain the intended (final) repositioning using the above described method, the code needs to be traversed backward to take into account transitivity of the avconds, and identify the program points (locations) to reposition the avconds. To avoid these backward traversals, we design an algorithm that automatically computes locations and conditions along with the avconds, so that the killed avconds can be directly repositioned using these values (and without performing backward code traversals). This algorithm is discussed in the next section.

3.5.3 Automated Computation of Repositioning Locations and Conditions

To eliminate the need to traverse the code backward, while computing the repositioning location for each of the killed avconds at any program point, we compute the repositioning location and condition for each avcond computed at any program point. Therefore, any avcond killed at a program point can be repositioned using the values computed for it. For this purpose, we design an algorithm that computes the following for each of the avconds, say c, computed at any program point p,

- The program point p_r that dominates p, and c is available at p through p_r directly or transitively: We refer to this program point p_r as *repositioning location* of c.

- The condition c_r available at p_r such that c_r reaches to p as c, directly or transitively, (i.e., c is a transformed version of c_r at p_r): we refer to this c_r at p_r as the *repositioning condition* of c.

Thus, due to the transitivity in avconds computation, we compute the repositioning location p_r and condition c_r for every avcond identified at any program point. These values are used later to implement the final repositioning. We stress that,

- An avcond computed at any program point has exactly one repositioning location p_r and one repositioning condition c_r associated with it (computed for it).
- Computing repositioning condition c_r for an avcond c is not required if avconds are not to be computed transitively (because in this case, c_r is same as c).
- The associated values, c_r and p_r, for an avcond are computed based on the conditions hoisted and their locations in the intermediate repositioning, rather than the original alarms input for repositioning and their locations.

Below we describe an algorithm that computes the repositioning location and condition of every avcond. We call this algorithm *avconds analysis*.

3.5.3.1 Notations

Let P be the set of all program points and C be the set of all conditions that can be formed using program variables, constants, and arithmetic and logical operators. Let function $\mathrm{f} : \mathrm{C} \mapsto \mathrm{C} \times \mathrm{P}$, map an avcond $c \in \mathrm{C}$ to its associated repositioning condition $c_r \in \mathrm{C}$ and location $p_r \in \mathrm{P}$. We write the condition c with the associated values as tuple $\langle c, c_r, p_r \rangle$. Thus, the values computed by avconds analysis at a program point are a subset of L_f, where $\mathrm{L}_f = \{\langle c, c', q \rangle \mid c \in \mathrm{C},\ \mathrm{f}(c) = \langle c', q \rangle\}$. Let *init* $= \{\langle c, \top_p, c \rangle \mid c \in \mathrm{C}\}$, where $\top_p$ is an artificial program point.

For a given set $S \subseteq \mathrm{L}_f$ we define the following function: *condsIn*$(S) = \{c \mid \langle c, c_r, p_r \rangle \in S\}$. That is, *condsIn*$(S)$ returns all avconds in S. We define the following functions for a given set $S \subseteq \mathrm{L}_f$, and $c \in$ *condsIn*(S).

- *repCond*(c, S) returns the repositioning condition associated with c, i.e., it returns c_r where $\langle c, c_r, p_r \rangle \in S$.
- *repLoc*(c, S) returns the repositioning location associated with c, i.e., it returns p_r where $\langle c, c_r, p_r \rangle \in S$.

Note that avconds computed by this analysis form a partially ordered set. However, the set of program points that can be computed for any avcond as its repositioning location does not form a partially ordered set. The relation among the program points—that can be computed for an avcond—is *successor of* which is not antisymmetric. As a result, a partial order that is reflexive, antisymmetric, and transitive, cannot be identified for the set of values computed by this analysis: avconds with the intended repositioning locations and conditions. Therefore, we do not call this analysis a data flow analysis, instead call it an algorithm. Although these values do not form a lattice, this analysis/algorithm still serves the purpose of computation of avconds with the intended values (discussed next).

Let $m, n \in \mathrm{N}$; $c, c', c_r \in \mathrm{C}$; $p_r \in \mathrm{P}$; and $X, Y \subseteq \mathrm{L}_f$.

$$\mathit{AvIn}_n = \begin{cases} \emptyset & n \text{ is } \textit{Start} \text{ node} \\ \prod\limits_{m \in \mathit{pred}(n)}^{n}{}_F \ \mathit{AvOut}_m & \text{otherwise} \end{cases} \tag{3.19}$$

$$\mathit{AvOut}_n = \mathit{Gen}_{\mathit{exit}(n)} \cup \mathit{AvOut'}_n \tag{3.20}$$

$$\mathit{Gen}_{\mathit{exit}(n)} = \{\ \langle c, c, \mathit{exit}(n)\rangle \mid c \in \mathit{Hoist}_{\mathit{exit}(n)}\ \} \tag{3.21}$$

$$\mathit{AvOut'}_n = (\mathit{AvIn'}_n \setminus \mathit{Kill}_n(\mathit{AvIn'}_n)) \cup \mathit{DepGen}(\mathit{AvIn'}_n)$$

$$\mathit{AvIn'}_n = \mathit{AvIn}_n \cup \mathit{Gen}_{\mathit{entry}(n)}(\mathit{AvIn}_n)$$

$$\mathit{Gen}_{\mathit{entry}(n)}(X) = \{\ \langle c, c, \mathit{entry}(n)\rangle \mid c \in \mathit{Hoist}_{\mathit{entry}(n)},\ \langle c', c_r, p_r\rangle \in X,\ \ c' \not\Rightarrow c\ \} \tag{3.22}$$

$$\mathit{Kill}_n(X) = \{\ \langle c, c_r, p_r\rangle \mid \langle c, c_r, p_r\rangle \in X,\ n \text{ contains a definition of an operand of } c\ \} \tag{3.23}$$

$$\mathit{DepGen}_n(X) = \begin{cases} \left\{ \langle \mathit{postcond}(n, c), c_r, p_r\rangle \;\middle|\; \begin{array}{c} \langle c, c_r, p_r\rangle \in X, \\ \mathit{postcond}(n, c) \text{ is computable} \end{array} \right\} & n \text{ is an assignment node} \\ \emptyset & \text{otherwise} \end{cases} \tag{3.24}$$

$$X\ {}^{n}\sqcap_F Y = \bigcup_{c \in \left(\mathit{condsIn}(X)\ \cap \mathit{condsIn}(Y)\right)} \{\ \mathit{mergeInfo}(c, \mathit{entry}(n), X, Y)\ \} \tag{3.25}$$

$$\mathit{mergeInfo}(c, p_m, X, Y) = \begin{cases} \langle c, c_r, p_r\rangle & \langle c, c_r, p_r\rangle \in X,\ \langle c, c'_r, p'_r\rangle \in Y,\ p'_r = \top_p \\ \langle c, c'_r, p'_r\rangle & \langle c, c_r, p_r\rangle \in X,\ \langle c, c'_r, p'_r\rangle \in Y,\ p_r = \top_p \\ \langle c, c, p_m\rangle & \langle c, c_r, p_r\rangle \in X,\ \langle c, c'_r, p'_r\rangle \in Y,\ p_r \neq p'_r \\ \langle c, c_r, p_r\rangle & \langle c, c_r, p_r\rangle \in X,\ \langle c, c'_r, p'_r\rangle \in Y,\ p_r = p'_r \text{ (and } c_r = c'_r\text{)} \end{cases} \tag{3.26}$$

Figure 3.6: Computation of repositioning locations and conditions for avconds.

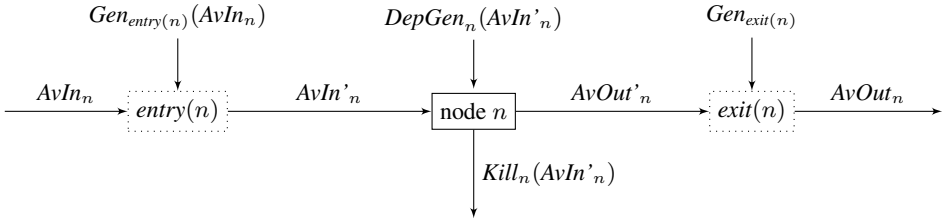


Figure 3.7: Processing of a node during avconds analysis

3.5.3.2 Computing Repositioning Locations using Avconds Analysis

Avconds analysis computes subsets of L_f flow-sensitively at every program point $p \in \mathrm{P}$. We initialize the values at every program point with *init*. Then we apply the equations shown in Figure 3.6, until the fixed point is computed (similar to the maximum fixed point solution computed by a data flow analysis). The equations transitively compute avconds in an intraprocedural setting, along with their repositioning locations and conditions. AvIn_n and AvOut_n denote avconds computed with their associated values, respectively, at the entry and exit of a node n (Equations 3.19

and 3.20). Figure 3.7 illustrates the computation of $AvOut_n$. Equations 3.21 and 3.22 indicate that an avcond is generated for every condition hoisted in the intermediate repositioning. At any program point, the avconds computed by this analysis are same as the avconds computed by the rel-alarms analysis in Section 3.5.2.1, with the only difference in the values associated with them. When an alarm condition is generated as an avcond at a program point, the same program point and the same condition is associated as the repositioning location and condition of the avcond (Equations 3.21 and 3.22).

Now, we describe the computation of repositioning location p_r and c_r of each avcond c that flows-in at a meet point, i.e., merging of associated values of c. At a meet point $entry(n)$, the computation of avconds with their associated values is shown by Equation 3.25. During this computation, the repositioning location and condition of an avcond c are respectively updated to $entry(n)$ and c, only if the repositioning locations of c flowing-in via the two different paths at the meet point are different and none of the two locations is $\top_p$. When one of the repositioning locations is $\top_p$, the other repositioning location is associated with the avcond. In the other case, i.e., when the repositioning locations of c flowing-in via the two different paths at a meet point are same, the values associated with c remain unchanged. In this case, the values remain unchanged, because, when the repositioning locations of c flowing-in via the two different paths at a meet point are same, the repositioning conditions of c flowing-in via the two different paths are also same. Hence, it can be seen that the computation at a meet point is independent of the repositioning conditions of avconds: $c_r = c'_r$ in Equation 3.25 is shown only for completeness.

Note that, the updates to the repositioning location and condition of an avcond c is only when c is generated as an avcond or during merging of the values at a meet point (discussed above). Therefore, with these updates, we ensure that the associated repositioning location p_r of an avcond c satisfies the following: p_r dominates p, and c has availability through p_r to p directly or transitively, with c_r being the version of c at p_r. The values computed for every avcond at any program point being unique, the values computed by avconds analysis at any program point converge over multiple iterations, i.e., the fixed point gets computed. Therefore, the analysis/algorithm terminates. We have discussed rel-alarms analysis and avconds analysis separately only for simplicity of the analysis formalization and discussion. Both these analyses can be implemented together.

3.5.4 Algorithm for Improvement of the Intermediate Repositioning

Algorithm 1 presents steps to improve the intermediate repositioning, i.e., performing final repositioning of alarms using results of the avconds analysis described above (Section 3.5.3) and rel-alarms analysis (Section 3.5.2). The steps in the algorithm are described below.

3.5.4.1 Step 1 (Computation of Repositioned Conditions)

At each program point, we first *select* avconds whose associated values are used to perform final repositioning. For every avcond c selected, (1) the repositioning condition c_r of c is repositioned at the repositioning location p_r of c, and (2) a traceability link is created from the resulting repositioned condition to each of the rel-alarms of c. We select avconds for the final repositioning by processing every node n in the program using Equations 3.27 and 3.28.

$$Conds_{entry(n)} = condsIn\,(Kill_n\,(AvIn'_n)) \tag{3.27}$$

$$Conds_{exit(n)} = condsIn\,(AvOut_n) \setminus \bigcap_{s \in succ(n)} condsIn\,(AvIn_s) \tag{3.28}$$

Algorithm 1 Algorithm for Final Repositioning

```
global R_C;
procedure PERFORMREPOSITIONING
    R_C ← ∅;
    /* Step 1 - computation of repositioned conditions (Section 3.5.4.1) */
    for each node n ∈ 𝒩 do
        REPOSITION(Conds_entry(n), AvIn'_n, fwdIn'_n);
        REPOSITION(Conds_exit(n), AvOut_n, fwdOut_n);
    end for
    /* Special case for the program End node */
    REPOSITION(condsIn(entry(End)), AvIn_End, fwdIn_End);

    /* Step 2 - simplification of the repositioned conditions (Section 3.5.4.2) */
    R_S ← ∅;
    for each point p ∈ P do
        C ← {⟨c, p⟩ | ⟨c, p⟩ ∈ R_C};
        R_S ← R_S ∪ SIMPLIFYCONDS(C);
    end for

    /* Step 3 - clustering of the repositioned conditions (Section 3.5.4.3) */
    R_E ← DISCARDFOLLOWERS(R_S);

    /* Step 4 - postprocessing for fallback (Section 3.5.4.4) */
    R_f ← PERFORMFALLBACK(R_E, Φ);   /* Algorithm 2 */

    return R_f;      /* the final repositioned alarms */
end procedure

procedure REPOSITION(C, X, Y)
    for each condition c ∈ C do
        r ← ⟨REPCOND(c, X), REPLOC(c, X)⟩;   /* Creation of new repositioned alarms */
        R_C ← R_C ∪ {r};
        for each φ ∈ ALARMSOF(c, Y) do
            CREATELINK(r, φ);   /* Add a traceability link from r to φ (Section 3.5.4.1) */
        end for
    end for
end procedure
```

These equations (3.27 and 3.28) respectively compute the avconds that are no longer *available* at any program point immediately after the entry and exit of a node n. These equations are on similar lines of the two cases in Section 3.4.2. The processing of every node through the two equations ensures the following: (1) each avcond c generated at a program point p gets selected for repositioning along every path starting at p and ending at the program exit except when it transitively results into some other avcond, and (2) the selection along any such path is only once and it occurs at the last program point on the path where c is available.

As a special case, we select every condition $c \in$ *condsIn*(*entry*(*End*)) for the repositioning, because a few avconds like $n \neq 0$ in Figure 3.1 can reach the program end point, but not get computed by either of the equations 3.27 and 3.28 for any program point.

In Algorithm 1, we use $\langle c, p \rangle$ to denote a repositioned condition (alarm) resulting after repositioning of condition c at point p. Note that the algorithm uses rel-alarms of every selected avcond to create traceability links from the resulting repositioned condition. In Algorithm 1, we assume that the call to function *createLink*(r, ϕ) creates a traceability link from a given repositioned alarm r to an original alarm ϕ. A map is used to maintain traceability links from a repositioned alarm to its corresponding original alarms.

Below, we describe repositioning of the example alarms shown in Figure 3.1, obtained after Step 1. This repositioning is obtained based on avconds computed from the hoisted conditions in the intermediate repositioning (described in Section 3.4.2.6).

1. $i \geq 0$ && $i \leq 4$ is repositioned at *entry*(n_{11}) both A_{17} and A_{19} as its rel-alarms.
2. $n \neq 0$ is repositioned at *entry*(n_{32}) with D_{27} and D_{29} as its rel-alarms.
3. $j \geq 0$ && $j \leq 4$ is repositioned at *exit*(n_{34}) with A_{37} as its rel-alarm.
4. $j \geq -1$ && $j \leq 3$ is repositioned at *exit*(n_{34}) with A_{39} as its rel-alarm.

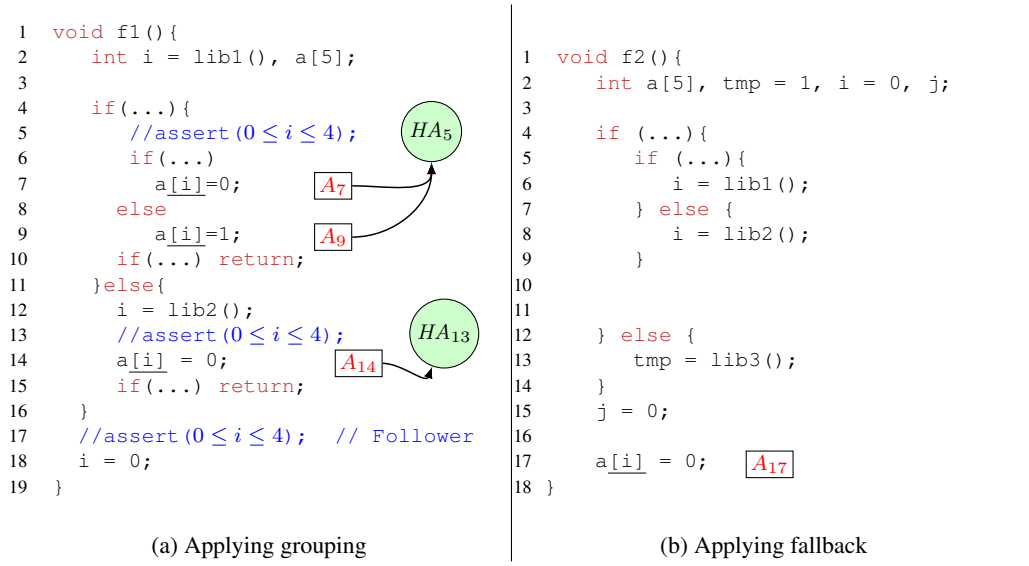
Note that, in each of the cases (1) and (2) above, the repositioned location is different from the locations of hoisted conditions from which the repositioned avcond is computed. However, in each of the cases (3) and (4) above, the repositioned location is same as the location of hoisted condition from which the repositioned avcond is computed.

3.5.4.2 Step 2 (Simplification of the Repositioned Conditions)

Next every program point is processed to simplify the conditions repositioned at that point. The simplification is performed on conjunction of the conditions that are repositioned at the same point and involve checking values of the same expression. The traceability links for a condition resulting after the simplification are obtained by merging traceability links of the conditions that got simplified. For example, after this simplification step, the two conditions $j \geq 0$ && $j \leq 4$ and $j \geq -1$ && $j \leq 3$ repositioned at *exit*(n_{34}) (Section 3.5.4.1) result in the simplified repositioned condition $j \geq 0$ && $j \leq 3$ at the same program point, with its traceability links to both A_{37} and A_{39}. In Algorithm 1, we assume that the function *simplifyConds(C)* accepts repositioned conditions C to be simplified and returns the conditions after their simplification. Moreover, we assume that it accordingly updates traceability links of the conditions.

3.5.4.3 Step 3 (Clustering of the Repositioned Conditions)

The repositioning resulting after the previous simplification step may have some redundancy, because some of the repositioned conditions can be follower alarms in the presence of the other repositioned conditions. As an example, consider the code in Figure 3.8a that has three AIOB alarms reported at lines 7, 9, and 14. Following we describe the repositioned conditions obtained after applying Step 2. The intermediate repositioning of these alarms results in hoisting condition $0 \leq i \leq 4$ at *entry*(n_6), and *exit*(n_{12}), i.e., the intermediate repositioning contains two hoisted conditions at *entry*(n_6), and *exit*(n_{12}). The repositioned conditions resulting after the improvement of the intermediate repositioning (i.e., Step 1) are at *entry*(n_6), *exit*(n_{12}), and *entry*(n_{18}). These three repositioned conditions are shown as assertions at lines 5, 13, and 17, respectively.



(a) Applying grouping (b) Applying fallback

Figure 3.8: Examples to illustrate postprocessing of the repositioned conditions.

The condition $0 \leq i \leq 4$ is repositioned at *entry*(n_{18}) (line 17) because the same condition is avcond at the exit of the program, with $0 \leq i \leq 4$ as its repositioning condition and *entry*(n_{18}) as its repositioning location. The repositioning of the condition at *entry*(n_{18}) denotes sinking of the two hoisted conditions (present in the intermediate repositioning) to the entry of n_{18}. The traceability links of this repositioned condition are to all the three original alarms.

Applying the simplification step (Step 2) to the three repositioned conditions does not reduce their number. Therefore, repositioning of the original three alarms (shown at lines 7, 9, and 14) does not reduce the number of alarms.

Observe that the condition repositioned at *entry*(n_{18}) (line 17) is redundant in presence of the two other repositioned conditions: the other two repositioned conditions act as *dominant alarms* for this condition. To improve the repositioning by eliminating such redundancy, (1) we postprocess the repositioned conditions by applying the clustering techniques [124, 150, 223], and (2) discard the repositioned conditions that are identified as *followers*. Applying this postprocessing step to the three repositioned conditions discards the redundant repositioned condition (line 17), and reduces the number of alarms by one. The final two repositioned conditions reported to the user are denoted using circles.

When a repositioned condition is discarded on finding it as a follower, its traceability links are transferred to its corresponding dominant repositioned conditions. In Algorithm 1, we assume that the function *discardFollowers*(R_S) performs clustering of given set of alarms R_S, and returns only the dominant alarms. Moreover, we assume that the function accordingly transfers traceability links of the follower alarms to their respective dominant alarms.

3.5.4.4 Step 4 (Postprocessing for Fallback)

In certain cases, the repositioning obtained after Step 3 may increase the number of alarms. This occurs due to (1) the two repositioning goals impacting each other (as illustrated below in certain scenarios), or (2) transitive computation of the antconds and avconds.

Algorithm 2 Algorithm for Repositioning Fallback Approach

Input: Set of original alarms Φ, set of repositioned alarms R
Output: Set of repositioned alarms R_f after fallback

```
procedure PERFORMFALLBACK(R, Φ)
    origAlarms ← Φ;
    R_f ← R;
    while origAlarms ≠ ∅ do
        φ' ← CHOOSE(Φ);
        Φ ← Φ \ {φ'};
        workingSet ← {φ'};
        origAlarmsSet ← ∅;
        reposAlarmsSet ← ∅;
        visitedOrigAlarms ← ∅;
        while workingSet ≠ ∅ do
            φ ← CHOOSE(workingSet);
            workingSet ← workingSet \ {φ};
            origAlarmsSet ← origAlarmsSet ∪ {φ};
            visitedOrigAlarms ← visitedOrigAlarms ∪ {φ};
            for each r ∈ GETRELREPOSALARMS(φ) do
                reposAlarmsSet ← reposAlarmsSet ∪ {r};
                for each φ' ∈ GETRELORIGALARMS(r) do
                    if φ' ∉ visitedOrigAlarms then
                        workingSet ← workingSet ∪ {φ'};
                    end if
                end for
            end for
        end while

        if |reposAlarmsSet| > |origAlarmsSet| then
            R_f ← (R_f \ reposAlarmsSet) ∪ origAlarmsSet;
        end if
        origAlarms ← origAlarms \ origAlarmsSet;
    end while
    return R_f;      /* Set of final repositioned alarms */
end procedure
```

Consider the code example in Figure 3.8b. This example is crafted to illustrate impact of the two repositioning goals on each other. The example has one alarm A_{17}. For this example, during the intermediate repositioning, the alarm's condition, $i \geq 0$ && $i \leq 4$, gets hoisted at *exit*(n_6) and *exit*(n_8). However, the condition hoisted at *entry*(n_{13}) gets discarded via Equation 3.11: the condition always holds at *entry*(n_{13}) because value of i is 0 at line 13 due to the assignment at line 2. During an avconds analysis, due to this discarding the condition is not avcond at *entry*(n_{13}), and hence at *entry*(n_{15}). Thus, repositioning of A_{17} results in two conditions repositioned at *exit*(n_6) and *exit*(n_8) (the repositioned conditions are not shown in Figure 3.8b). That is, for this example, the repositioning increases the number of alarms, from one to two.

We discard redundant conditions during the intermediate repositioning to report the alarms closer to their cause points. However, for this example, the discarding results in creating more repositioned alarms than the original alarms. In the absence of this discarding, the condition gets repositioned only at *entry*(n_{15}), without increasing the number of alarms. This indicates the two repositioning goals impact each other, i.e., reporting alarms closer to their causes might increase the number of alarms.

Hence, we postprocess the repositioned conditions resulting after Step 3. In this postprocessing step, we identify situations in which the number of alarms increases, and revert the repositioning in these situations, i.e., we report original alarms instead of the repositioned ones. We call this approach *fallback*. Algorithm 2 presents the fallback approach, where it, for simplicity, assumes the following two functions.

- *getRelOrigAlarms(r)* returns set of rel-alarms associated with a given repositioned alarm condition r.
- *getRelReposAlarms(ϕ)* returns set of repositioned alarms for which ϕ is one of their rel-alarms.

As shown by our experimental evaluation, discussed in the next section, fallback is rarely required[6].

3.5.4.5 Final Repositioning Example:

Applying Algorithm 1 to the alarms in Figure 3.1 results in the following final repositioning. For this example, the postprocessing steps 3 and 4 do not change (improve) the repositioning obtained after Step 2.

1. $i \geq 0$ && $i \leq 4$ is repositioned at *entry*(n_{11}) with its traceability links to A_{17} and A_{19}, where the repositioning is identified by Equation 3.28 when applied to the *exit*(n_{11}).
2. $n \neq 0$ is repositioned at *entry*(n_{32}) with its traceability links to D_{27} and D_{29}, where the repositioning is identified when *entry*(End) is processed as the special case.
3. $j \geq 0$ && $j \leq 3$ is repositioned at *exit*(n_{34}) with its traceability links to A_{37} and A_{39}.

3.5.4.6 Properties of Repositioning Algorithm 1

In this section we prove properties of Algorithm 1.

Theorem 3.5.1. *Given a set of alarms Φ, the repositioning of Φ resulting from Algorithm 1 is* safe.

Proof. Recall that repositioning of a set of original alarms S is safe if the following holds: each of the resulting repositioned alarms is a false positive *if and only if* its corresponding original alarms in S are false positives (Definition 3.3.3). Let Φ_R be a set of alarms resulting after repositioning of Φ by Algorithm 1. Proving repositioning of Φ obtained through Algorithm 1 is safe requires the following two cases to be proved.

1. For every original alarm $\phi_p \in \Phi$ detecting an error at p, there exists an alarm $\phi'_q \in \Phi_R$ denoting an error at q and detecting the same error at p, and vice versa.

[6]In fact, there existed only 20 instances that required fallback during repositioning of 33,162 alarms in our experimentation.

2. When every original alarm $\phi_p \in \Phi$ is false, all alarms in Φ_R are also false, and vice versa.

In the repositioning resulting due to Algorithm 1, the following holds true.

(A) Corresponding to every alarm $\phi_p \in \Phi$, there exists a repositioned alarm $\phi'_q \in \Phi_R$ along every path on which p appears either before or after p (Section 3.5.4.1), except the paths along which ϕ_p is always safe (due to discarding of redundant hoisting). Furthermore, $cond(\phi'_q)$ along each path is directly or transitively derived from $cond(\phi_p)$.

(B) Corresponding to every repositioned alarm $\phi'_q \in \Phi_R$, there exists an original ϕ_p alarm along every path on which q appears, and $cond(\phi')$ is directly or transitively derived from $cond(\phi)$. This follows from the fact that the repositioning location of an avcond is created either (i) from a location of hoisted condition in intermediate repositioning or (ii) during a meet operation (Equation 3.25). In the first case (i), the location of each hoisted condition in intermediate repositioning is such that there exists a related original alarm on every path passing through the location, because the hoisted conditions are computed based on antconds. In the second case (ii), when a meet operation is performed, the new repositioning location of an avcond is updated to the the meet point when the avcond has two different repositioning locations. Therefore, for the conditions repositioned at the meet point, there exists a related original alarm along every path that passes through the meet point.

Proving case 1 (Detection of an error):
Due to (A), $\forall \phi_p \in \Phi : (!cond(\phi_p)) \Rightarrow \exists\ \phi' \in \Phi_R, (!cond(\phi'))$.
Due to (B), $\forall \phi' \in \Phi_R : (!cond(\phi')) \Rightarrow \exists\ \phi \in \Phi, (!cond(\phi))$.
Proving case 2 (Alarms are false positives):
Due to (A), $\forall\ \phi_p \in \Phi, (cond(\phi_p)) \Rightarrow \forall\ \phi' \in \Phi_R, (cond(\phi'))$.
Due to (B), $\forall\ \phi' \in \Phi_R, (cond(\phi')) \Rightarrow \forall\ \phi \in \Phi, (cond(\phi))$. □

Theorem 3.5.2. *For any given set of alarms, Algorithm 1 always terminates.*

Proof. The termination of Algorithm 1 is proved by proving termination of each of the steps in the algorithm.
Step 1: This step computes final repositioning locations using the results of antconds analysis (Section 3.4.1), rel-alarms analysis (Section 3.5.2), and avconds analysis (Section 3.5.3). As the lattices of antconds and rel-alarms analyses have finite descending chains (Section 3.4.1.2), and the data flow equations in Figures 3.2, and 3.4 are *monotonic* (Section 3.4.1.3), both the analyses terminate [100, 163]. As discussed in Section 3.5.3.2, avconds analysis always terminates. Since this step processes every program point corresponding to a node in finite set of nodes N, this step always terminates.
Step 2: This step terminates because it iterates over all the program points to simplify conjunction of the repositioned conditions.
Step 3: In this step, state-of-the-art clustering techniques [71, 124, 150, 223] are applied to postprocess the repositioned conditions and these techniques are always terminating.
Step 4: Algorithm 2 presents identification of the situations where repositioning increases the alarms count and reverting repositioning in those situations. It is described in detail in Section 3.5.4.4. The repositioned and original alarms processed by Algorithm 2 to perform fallback are finite. In each iteration of the outer loop, at least one original alarm is processed and removed from the set $origAlarms$. When all the original alarms in the set are processed (and removed from the set), Algorithm 2 terminates. Therefore, this step (Step 4) of Algorithm 1 also terminates. □

Theorem 3.5.3. *For any given set of alarms, Algorithm 1 never increases the number of alarms after repositioning.*

Proof. In Step 4 (Section 3.5.4.4), Algorithm 2 identifies situations for repositioning fallback: instances where the number of repositioned alarms is higher than the number of their corresponding original alarms (rel-alarms) and reporting the original alarms instead of the repositioned ones. In the other situations, the number of repositioned alarms is equal to or lesser than the number of their corresponding original alarms. Thus, Algorithm 1 never increases the number of alarms after repositioning. □

Theorem 3.5.4. *For a given set of dominant alarms Φ, Algorithm 1 performs sinking of an alarm $\phi \in \Phi$ only if there is an impending reduction in number of overall alarms.*

Proof. The sinking of alarms occurs during avconds analysis when two or more same alarm conditions but from different alarms meet for the first time. This sinking is denoted by Equation 3.25, where the two alarms are merged together to depict a single alarm. Such merging (sinking) of alarms contribute to reducing overall alarms count. For a single alarm, such sinking is not performed. Although such merging can result in repositioning conditions that are redundant (Section 3.5.4.3). However, such redundant alarms resulting due to sinking are discarded via the postprocessing Step 3. Thus, Algorithm 1 performs sinking of an alarm only if there is an impending reduction in number of overall alarms. □

3.6 Empirical Evaluation

In this section, we evaluate the alarms repositioning technique (Algorithm 1) that we presented to reduce the number of alarms. We evaluate the technique using alarms generated by a commercial static analysis tool on real-world applications. In the evaluation, we measure the reduction in alarms that we obtain using the technique.

3.6.1 Experimental Setup

3.6.1.1 Implementation

We implemented alarms repositioning on top of analysis framework of our commercial static analysis tool, TCS ECA [197]. The analysis framework supports analysis of C programs, and allows to implement data flow analyses using function summaries. We implemented antconds analysis (Section 3.4.1), rel-alarms analysis (Section 3.5.2), and avconds analysis (Section 3.5.3), to compute the conditions transitively. In each of these analyses, we computed the conditions interprocedurally, by solving the data flow analyses in bottom-up order only. In the antconds analysis, for every function we propagated the conditions anticipable at its entry to its caller, only if that function is called from a single place. In the avconds analysis, all the conditions available at the exit of each function are propagated to all the caller(s) of that function irrespective of the call invocations of the function. This implementation may result in repositioning an original alarm at multiple locations, and for such cases we resort to the fallback approach. We implemented computation of weakest precondition (resp. strongest postcondition) required in antconds (resp. avconds) analysis only for the assignment nodes of simple types, e.g., $x = y$; $x = y \pm c$; and $x = c \pm y$; where c is a constant. This limited implementation does not impact correctness of the repositioning technique, however due to this limitation the technique can fail to group alarms which could be grouped otherwise.

Table 3.1: Experimental results showing reduction in number of alarms due to their repositioning. The columns *#Input*, *#Output*, and *%Reduction* respectively present the number of alarms before and after repositioning, and the percentage of alarms reduced. The column *RCs* presents the number of repositioned conditions identified as *followers* (Section 3.5.4.3), while the column *FBs* presents the number of instances where fallback gets applied (Section 3.5.4.4). The other columns (*Interprocedural* and *Reasons for Stopping Repos.*) are described in Section 3.6.3.

Application	Size (KLOC)	# Input	# Output	% Reduction	RCs	FBs	Timing Analysis (seconds)			Interprocedural		Reasons for Stopping Repos.		
							Orig. analysis	Repositioning	% Overhead	Mergings	Repos.	Func entry	Branches	Defs
acpid-1.0.8	1.7	5	4	20.00	0	0	4.5	2.6	57.0	0	0	1	2	1
spell-1.0	2.0	17	17	0.00	0	0	15.2	4.7	30.9	0	0	3	9	5
barcode-0.98	4.9	580	540	6.90	0	0	54.2	12.4	23.0	0	0	2	363	175
antiword-0.37	27.1	748	689	7.89	6	0	941.7	116.1	12.3	1	35	93	454	142
sudo-1.8.6	32.1	2548	2407	5.53	28	0	2618.9	451.5	17.2	30	54	303	971	1133
uucp-1.07	73.7	263	244	7.22	4	0	455.3	42.2	9.3	0	0	17	134	93
ffmpeg-0.4.8	83.7	18523	17557	5.22	169	12	2059.2	535.4	26.0	85	610	1252	10263	6042
sphinxbase-0.3	121.9	908	887	2.31	24	0	162.0	48.1	29.7	5	41	39	662	186
archimedes-0.7.0	0.8	2251	2146	4.66	6	0	27.4	8.6	31.5	1	15	5	1078	1063
polymorph-0.4.0	1.3	10	8	20.00	0	0	5.3	2.1	39.5	0	0	1	6	1
nlkain-1.3	2.5	89	88	1.12	0	0	5.0	1.9	37.2	0	0	0	59	29
stripcc-0.2.0	2.5	88	77	12.50	8	0	17.5	2.8	16.1	0	0	2	49	26
ncompress-4.2.4	3.8	64	58	9.38	0	0	5.9	3.4	57.2	0	0	1	33	24
barcode-0.96	4.2	440	408	7.27	0	0	39.3	11.3	28.7	0	0	2	285	121
combine-0.3.3	10.0	454	407	10.35	9	0	46.7	12.5	26.8	0	2	9	295	103
gnuchess-5.05	10.6	1600	1503	6.06	40	0	86.4	19.9	23.0	10	38	90	782	631
industryApp 1	3.4	326	266	18.40	0	0	20.9	8.8	42.0	0	20	7	148	111
industryApp 2	18.0	163	162	0.61	1	0	44.4	11.4	25.8	0	5	14	112	36
industryApp 3	18.1	1111	1007	9.36	1	0	72.6	21.8	30.0	0	31	16	800	191
industryApp 4	30.9	2974	2541	14.56	1	11	1253.5	76.4	6.1	44	592	167	1675	699
Total		33162	31016	6.47	297	23	7935.9	1393.9	17.6	176	1443	2024	18180	10812

3.6.1.2 Selection of Applications and Alarms

For evaluation purpose, we selected 20 applications (Table 3.1): 16 open source and four industry applications. All these applications were analyzed using TCS ECA on a machine with i7 2.5GHz processor and 16GB RAM. The open source applications are selected from the benchmarks used for evaluating the clustering techniques [124, 223]: the first eight applications are from the study performed by Zhang et al. [223] and the next eight are from the study performed by Lee et al. [124]. The remaining benchmarks from these studies either were not available or could not be compiled or analyzed using TCS ECA. The industry applications selected are embedded systems from the automotive domain. All the applications selected are coded in C.

We selected alarms corresponding to four commonly checked verification properties: division by zero (DZ), array index out of bounds (AIOB), integer overflow underflow (OFUF), and uninitialized variables (UIV). The alarms selected were postprocessed using state-of-the-art alarms clustering techniques [124, 150, 223], and we considered only the dominant alarms as input to the repositioning. We performed clustering of alarms before their repositioning, because as indicated in Sections 3.1 and 3.2; we aim at overcoming the limitations of grouping techniques. Due to possible side effects caused by function calls, alarms having *function calls*

Table 3.2: Reduction in alarms error category-wise.

	#Input	#Output	%Reduction
AIOB	3464	3221	7.02
DZ	985	975	1.02
OFUF	24843	23564	5.15
UIV	3914	3607	7.84

in their alarm conditions are excluded from grouping [150]. Thus, we also have excluded them from repositioning.

3.6.2 Evaluation Results

Table 3.1 presents the number of alarms before and after repositioning, and the percentage of alarms reduced: columns *#Input*, *#Output*, and *%Reduction* respectively. The percentage of reduced alarms ranges between 0 and 20%, with median reduction of 7.25% and average reduction of 6.47%. The average reduction on open source applications is 5.41% as compared to 13.07% on the industry applications. We performed a study to understand the reasons for higher reduction on the industry applications. However, results of the study were inconclusive.

Table 3.1 also details the improvements resulting from the postprocessing of repositioned conditions (steps 3 and 4 in Section 3.5.4). Column *RCs* of Table 3.1 presents the number of repositioned conditions identified as *followers*, i.e., redundant conditions, by the clustering technique in Step 3 (Section 3.5.4.3). It indicates that around 1% of the repositioned conditions computed by Step 2 are identified as redundant by Step 3. Column *FBs* presents the number of instances, 23, where fallback got applied (Section 3.5.4.4). This indicates that the fallback gets rarely applied in practice. Our manual analysis of these instances showed that (a) three instances were due to the kind of inter-procedural implementation we had for avconds/antconds computation; (b) 18 instances were because of the two repositioning goals impacting each other; and (c) the other 2 cases were due to computing the conditions transitively.

To compute performance overhead incurred by the repositioning, we compared the time for repositioning (column *Repositioning*) to the time for (1) analyzing the code for the categories selected and (2) clustering the TCS ECA-generated alarms (column *Orig. analysis*). On average, the repositioning added performance overhead of 17.6% while it reduced the alarms count by 6.47%.

To investigate which verification properties are benefited the most through repositioning, we performed evaluation by repositioning alarms in each category separately. The evaluation results in Table 3.2 shows that reduction percentages for AIOB, OFUF, and UIV are comparable and are lowest for DZ. Our analysis of the results for DZ showed that the division operations mostly appear in only one branch of an *if* condition. In such cases, repositioning is unable to merge such alarms.

3.6.3 Discussion and Future Work

The reduction in alarms due to repositioning, 7.25%, is on top of the alarms reduction obtained through the clustering techniques. Thus, the reduction indicates failure of the clustering techniques to merge those alarms. Recall that the repositioning technique also uses alarms clustering to identify and discard redundant repositioned conditions (Section 3.5.4.3). Thus, clustering and

repositioning techniques help each other when used to reduce the number of alarms. Furthermore, we observe that, if implication is handled during the computation of antconds and avconds (Equations 3.6 and 3.22), the presented repositioning subsumes clustering of alarms, and therefore clustering of original alarms can be skipped when their repositioning is performed.

We believe that inter-procedural hoisting of alarms can provide more benefits in manual inspection of alarms. The expected gain is due to eliminating code traversals from the functions of their original reporting to the functions having repositioned alarms. From the evaluation, we also see that while merging alarms originally reported in different functions is not frequent (column *Mergings*), repositioning alarms across the function boundaries is quite common (column *Repos.*).

The repositioning of alarms mainly comprises upward repositioning (hoisting), whereas the forward repositioning (sinking) is performed only when hoisting does not help to reduce their number. Our attempt to understand the *reasons for stopping the backward repositioning* showed the following.

1. For around 6% of the repositioned alarms, the backward repositioning stopped at the entry of a function as the function was called from more than one place (column *Func entry*);

2. For around 59% of the repositioned alarms, the backward repositioning stopped due to branching nodes: the repositioned alarm appears only in one branch of the *if* statement (column *Branches*);

3. For the other repositioned alarms (35%), the backward repositioning stopped due to definitions of a variable appearing in the alarm conditions (column *Defs*).

Majority of the upward repositioning of alarms is stopped in Case 2. We stop the upward repositioning of an alarm in this case, i.e., when the alarm appears in only one branch of an *if* statement, because the *if* statement can prevent the alarm from being an error. If the repositioning is continued beyond the *if* statement due to which the alarm is false positive, we cannot guarantee that the required repositioning constraint is met (whenever the repositioned alarm is false positive, its corresponding original alarms are also false positives, and vice versa). Repositioning an alarm further upward, i.e., beyond *if* statements that do not prevent it from being an error, can allow to merge it with another and reduce their number. To this end, in the next chapter (Chapter 4), we design a strategy to identify *if* statements that are *non-impacting* to the alarms, and therefore should not stop the repositioning process.

3.7 Related Work

As there exist six categories of approaches for alarms postprocessing, and multiple techniques in each category, we limit the comparisons of our technique to the approaches/techniques that are closely related. As discussed in Section 3.2, clustering of alarms based on similarity/correlations is the most closely related approach to repositioning. State-of-the-art clustering techniques [124, 150, 223] have helped to reduce the number of alarms significantly (34 to 60%). However, they fail to group alarms in certain cases (discussed in Section 3.2) and also report the alarms away from their causes. As discussed in Section 3.2, our approach to reposition alarms is motivated by the work by Gerhke et al. [71]. Their work aims to overcome limitations of alarms clustering techniques by repositioning alarms. The technique they use to reposition and group similar alarms sometimes ends up creating more alarms than the alarms input for repositioning. Moreover, the technique does not perform sinking of alarms, and does not maintain traceability

link(s) between a repositioned alarm and its corresponding original alarm(s). In the absence of these links, reviewer needs to perform additional code traversals but in forward direction to locate the original alarms corresponding to a repositioned alarm, when (1) the repositioned alarm denotes an error, and (2) a correction is needed at its corresponding original alarm program point(s). Our technique has been designed to overcome these limitations.

Cousot et al. [39] have proposed usage of necessary preconditions which are hoisted to the method entry, corresponding to the inevitable checks within a method. The conditions hoisting is used in the context of providing the preconditions required by the Design by Contract [145]. On similar lines, Das et al. [48] have proposed *angelic verification* technique for verification of open programs. This technique is intended to prune the alarms generated during verification of open programs with an unconstrained environment. The alarms repositioning technique is applicable to programs with both constrained and unconstrained environments.

Muske and Khedker [154] have proposed cause points analysis to handle alarms effectively during the manual inspections. In their approach, instead of alarms, ranked causes to the alarms are reported and user inputs are sought in several iterations to resolve the alarms. This approach does not reduce the number of alarms without user intervention.

We observe that alarms repositioning can be applied in conjunction with other alarms postprocessing techniques to complement each other, and also, we believe that the combinations will provide more benefits as compared to the benefits obtained by applying them individually. Benefits of such combinations should be subject of further studies.

3.8 Conclusion

In this chapter, we have proposed a novel alarms postprocessing technique intended mainly to reduce the number of alarms. We obtain a reduction by overcoming the limitation of the existing alarms clustering techniques (addressed by RQ 2) that fail to group similar alarms, e.g., when similar alarms appear in different branches of an *if* statement. The failure occurs because they report alarms at the program points where alarms are generated. The novelty of our repositioning approach consists in reporting those alarms at some other locations, which subsequently allows to reduce the number of alarms appearing in those limitation cases. Therefore we find that,

> Repositioning alarms to other locations than their original locations helps to overcome the limitation of state-of-the-art clustering techniques.

We have evaluated the technique using 33,162 alarms generated by our commercial static analysis tool, TCS ECA, on 20 open source and industry applications. The evaluation results indicate that,

> Repositioning of alarms helps to reduce the number of alarms up to 20%, with median reduction of 7.25%.

In addition to the primary goal of reducing the number of alarms, the repositioning technique can be used to report alarms as close as possible to their cause points. Such reporting helps to reduce backward code traversals performed during the manual alarms analysis. We believe that the repositioning technique, being orthogonal to many of the existing approaches to postprocess alarms, can be applied in conjunction with those approaches.

Prior techniques [71, 124, 150, 223], which group similar alarms, have measured their effectiveness by measuring reduction in the number of alarms. Therefore, on similar lines, we measured effectiveness of the proposed repositioning technique by measuring reduction in the number of alarms. Evaluating its effectiveness in terms of actual benefit—reduction in manual inspection effort—due to reducing the number of alarms and backward code traversals is a part of future work. Such evaluations require performing a controlled study by involving multiple participants who are (experienced) users of static analysis tools: identifying such participants and involving them in a controlled study is difficult and costly.

In our evaluation we observed that, in majority of the cases, the hoisting repositioning of alarms is stopped due to the conservative assumption made about controlling conditions of the alarms when the alarms appear in only one branch of the *if* statements. Therefore, we aim to overcome this limitation. To this end, in the next chapter (Chapter 4), we design a strategy to identify controlling conditions of alarms, that do not impact the alarms. Considering the effect of those *non-impacting controlling conditions* can help to further reduce the number of alarms.

Chapter 4

NCD-based Repositioning of Alarms

Static analysis tools help to detect programming errors but generate a large number of alarms. In Chapter 3 we have proposed a technique to reduce the number of alarms by repositioning them. The repositioning technique replaces a group of similar alarms by a single repositioned alarm, e.g., when they belong to different branches of a conditional statement. *However, as observed in Section 3.6.3 the technique fails to merge similar alarms mainly when the immediately enclosing conditional statements of the alarms are different and not nested. This limitation is due to our conservative assumption that a conditional statement of an alarm may prevent the alarm from being an error. Our pilot study on 16 open source applications indicates that, as a result of this failure, 38% of the alarms obtained after repositioning are similar but not grouped together.*

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 alarms that are similar, and then reposition these similar alarms by considering the effect of their NCDs. We call this variant of repositioning* NCD-based repositioning. *Compared to the original repositioning technique, NCD-based repositioning allows to merge more similar alarms together and represent them by a small number of representative repositioned alarms. Thus, it can be expected to further reduce the number of alarms.*

To measure the reduction obtained, we evaluated NCD-based repositioning using 105,546 alarms generated on the 20 applications previously used during evaluation of the original repositioning technique (Section 3.6.1.2), and 12 additional industry applications. The evaluation results indicate that, compared to the original repositioning technique, NCD-based repositioning reduces the number of alarms respectively by up to 23.57%, 29.77%, and 36.09%. The median reductions respectively are 9.02%, 17.18%, and 28.61%.

4.1 Introduction

In this section, we describe the limitation of alarms repositioning technique (Section 3.6.3), that we address in this chapter and present overview of our solution. We begin by providing brief background to the alarms clustering and repositioning techniques.

4.1.1 Background

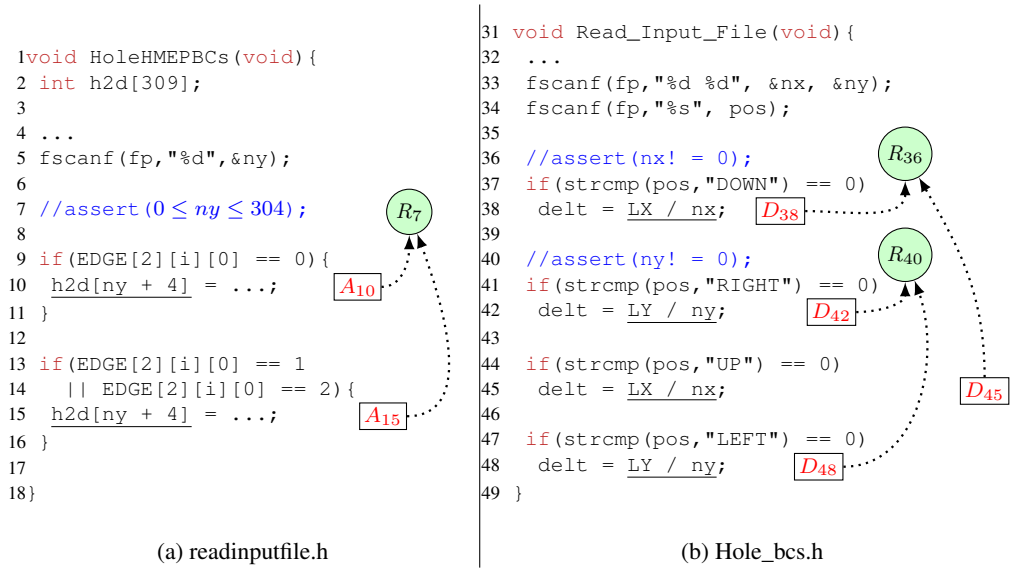
Static analysis tools help to automatically detect common programming errors like *division by zero* and *array index out of bounds* [12, 14, 21, 206] as well as to certify absence of such errors in safety-critical systems [24, 52, 112]. However, these tools report a large number of alarms that are warning messages notifying the tool-user about potential errors [54, 92, 139, 181, 186]. Partitioning alarms into true errors and false alarms (false positives) requires manual inspection [54, 120, 182]. 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 [20, 37, 92, 120].

Clustering of alarms, one of the six categories of approaches that we identify in the literature study (Chapter 2), is commonly used to reduce the number of alarms. State-of-the-art clustering techniques [71, 123, 150, 223] group similar alarms together such that (1) there are a 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 clustering.

The clustering techniques [71, 123, 150, 223] fail to group similar alarms which could be grouped together. To overcome the limitation of the techniques, we proposed repositioning of alarms (Chapter 3). Our technique repositions a group of similar alarms to a program point where they can be *safely replaced* by a single newly created representative alarm (called *repositioned alarm*). The alarms repositioning is safe and performed only if the following *repositioning criterion* is met—a repositioned alarm is a false positive if and only if its corresponding *original alarms* are all false positives. Thus, repositioned alarms act as dominant alarms for the original similar alarms that are replaced by them. Henceforth in this chapter, we call the technique proposed in Chapter 3 *original repositioning technique* (ORT).

4.1.2 The Problem

In our evaluation of ORT (Section 3.6.3), we observed that the technique fails to reposition and merge similar alarms when their immediately enclosing conditional statements are (1) different, and (2) non nested (i.e., immediately enclosing conditional statement of one alarm is not a conditional statement of another alarm's immediately enclosing conditional statement). As a consequence, in this case, the technique does not reduce the number of similar alarms. We call these cases *repositioning limitation scenarios*. We illustrate this limitation using the alarms (rectangles) shown in Figure 4.1. The code is excerpted from archimedes-0.7.0. The two code examples shown in Figures 4.1a and 4.1b are independent of each other. Analyzing the code in Figure 4.1a (resp. Figure 4.1b) 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 clustering techniques does not reduce their number.



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

Figure 4.1: Examples of alarms to illustrate their NCD-based repositioning.

Among the six alarms shown in Figure 4.1, there exist three groups of similar alarms: A_{10} and A_{15}, D_{38} and D_{45}, and D_{42} and D_{48}. ORT cannot determine whether the control dependencies[1] (i.e. the enclosing conditional statements) of these alarms *can prevent* the alarms from being an error. Thus, it 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, ORT 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 to any 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 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 4.1. As a result, the technique does not reduce the number of alarms shown in Figure 4.1.

We find that the above assumption about the control dependencies of the alarms' program points limits the reduction achieved by ORT, 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 4.1 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 (see Section 4.4.1).

Our pilot study using 64,779 alarms generated on 16 open source applications (Section 4.3) indicates that 38% of the alarms reported after their repositioning are still similar and appear in

[1]A control dependency of a program point p is a conditional edge in the control flow graph [5], which decides whether p is to be reached or not (see Section 4.2.1).

the repositioning limitation scenarios. These results suggest the scope for improvement. To this end, we ask the following research question.

> **RQ 3:** How can we improve the reduction in the number of alarms obtained by repositioning them?

4.1.3 Overview of 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[2] 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 computation of NCDs to similar alarms only. Since determining whether a control dependency is an NCD is undecidable (Section 4.4.2), we compute NCDs of similar alarms approximately. The NCDs computed are subsequently used to reposition the similar alarms. Therefore, this variant of repositioning, that we call *NCD-based repositioning*, allows to reposition more similar alarms together and replace them by fewer repositioned (dominant) alarms than ORT. For example, our approach to compute NCDs, identifies the control dependencies of the alarms shown in Figure 4.1 as NCDs. Repositioning each group of similar alarms using the NCDs allows to replace the group by a newly created dominant alarm (shown using 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 previously used as benchmarks for evaluating ORT (Section 3.6.1.2); (ii) 4 industry C applications that were previously used as benchmarks to evaluate ORT (Section 3.6.1.2), and 7 additional industry C applications; and (iii) 5 industry COBOL applications. The evaluation results indicate that, compared to ORT, 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 presented in this chapter.

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 technique using 105,546 alarms on 16 open source and 16 industry applications.

Chapter Outline Section 4.2 presents terms and notations that we use throughout this chapter and later in Chapter 6. Section 4.3 describes the pilot study. Section 4.4 describes the notion of NCDs and NCD-based repositioning. Section 4.5 presents a technique/algorithm to implement NCD-based repositioning. Section 4.6 describes our empirical evaluation. Section 4.7 presents related work, and Section 4.8 concludes.

[2] A conditional edge e is called a transitive control dependency of a point if the edge belongs to transitive closure of control dependencies of that program point (see Section 4.2.1).

```
1   void f1(int p, int q){
2     int t, arr[10], i;
3
4     i = readInt();
5
6     if(p == 1)
7       i = 0;
8
9     if(q == 100)
10      if(p == 1)
11        arr[i] = 0;      A11

12    if(q == 5){
13      arr[i] = 1;        A13
14      print(20 / i);           D14
15    }
16
17    if(q == 5)
18      t = arr[i];        A18
19
20    q = lib();
21    if(q == 5)
22      t = 0;
23 }
```

Figure 4.2: Examples to illustrate ICDs and NCDs of alarms.

4.2 Terms and Notations

We use the same terms and notations described in Section 3.2.1 for control flow graph and program points. Below we describe a few additional terms and notations that we use in this chapter and Chapter 6. Examples of these terms and notations are provided by referring to Figure 4.2.

4.2.1 Data and Control Dependencies

Let $d_x : x = y$ be a definition of x at a program point d_x. A definition $d_x : x = y$ of x is said to be a *reaching definition* of x at a program point p if there exists a path from the program entry to p such that the point d_x is on this path and x is not redefined along the path between d_x and p [100, 163]. For example, $i = readInt()$ (at line 4) and $i = 0$ (at line 7) are reaching definitions of i at line11. A variable v at a program point p is said to be *data dependent* on a definition d_v of v, if d_v is a reaching definition of v at p. Data dependencies of a variable v are the definitions on which v is data dependent. For example, $i = readInt()$ (at line 4) and $i = 0$ (at line 7) are data dependencies of i at line11.

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 [44, 64]. For example, n_7 is control dependent on edge $n_6 \rightarrow n_7$. 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. For example, edge $n_6 \rightarrow n_7$ is a control dependency of n_7, *entry*(n_7), and *exit*(n_7). A conditional edge e is called *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. For example, $(n_9 \rightarrow n_{10}) \rightsquigarrow n_{11}$, and $(n_9 \rightarrow n_{10}) \rightsquigarrow$ *entry*(n_{11}), because it is a control dependency of n_{10} which is source node of the control dependency of n_{11}.

Definition 4.2.1 (Equivalent Conditions)**.** We say that 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 non-equivalent. ■

For example, (1) the conditions of $n_6 \rightarrow n_7$ and $n_{10} \rightarrow n_{11}$ are equivalent, and (2) the conditions of $n_{12} \rightarrow n_{13}$, $n_{17} \rightarrow n_{18}$, and $n_{21} \rightarrow n_{22}$ are also equivalent. Note that, as conditions of conditional expressions are logical formulas without considering the context of the program point where they are present, checking their equivalence is same as checking whether

two logical formulas are equivalent. Therefore, in the second example, the conditions of $n_{17} \rightarrow n_{18}$, and $n_{21} \rightarrow n_{22}$ are also equivalent even though the variable q can take different values at their corresponding program points.

Definition 4.2.2 (Condition-wise Equivalent Conditional Edges)**.** 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). ■

For example, in Figure 4.2, (1) $n_6 \rightarrow n_7$ and $n_{10} \rightarrow n_{11}$ are condition-wise equivalent, and (2) $n_{12} \rightarrow n_{13}$ and $n_{17} \rightarrow n_{18}$ are also condition-wise equivalent. Note that, although $n_{12} \rightarrow n_{13}$ and $n_{21} \rightarrow n_{22}$ have equivalent conditions, they are not condition-wise equivalent: the reaching definitions of the variable q in their conditions are different due to the assignment to q at line 20.

4.2.2 Static Analysis Alarms

Recall that we use *cond*(ϕ) to denote the *alarm condition* of an alarm ϕ, i.e., the check performed by the analysis tool for detecting an error (Section 3.2.2). The alarm condition holds *iff* the corresponding alarm is a false positive. For an evaluation of an alarm condition *cond*(ϕ), let the values of variables in the evaluation are denoted as a tuple. We use *safe values* (resp. *unsafe values*) to refer to the set of all possible tuples of values of the variable(s) in *cond*(ϕ), where the values in each tuple evaluates *cond*(ϕ) to true (resp. false).

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 newly created (repositioned) alarm at p with c as its alarm condition.

4.3 Pilot Study

Usefulness of the repositioning technique we will develop in this chapter is based on two assumptions: (1) large number of alarms resulting from ORT are similar, and (2) large percentage of these similar alarms appear in the limitation scenarios (Section 4.1). Indeed, if only few alarms are similar, or if the similarity is not related to the limitation scenarios, impact of the technique to be developed will be negligible. Hence, in this section we perform a preliminary study to measure (1) what percentage of alarms resulting after ORT are similar; and (2) what percentage of these similar alarms appear in the repositioning limitation scenarios. 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 for evaluating ORT (Section 3.6.1.2). We analyzed these applications using our static analysis tool, TCS ECA [197], for five verification properties: *division by zero*, *array index out of bounds (AIOB)*, *arithmetic overflow and underflow*, *dereference of a null pointer*, and *uninitialized variables*. As we aim to measure percentage of similar alarms resulting after applying ORT, the tool-generated alarms are postprocessed using the clustering techniques [124, 150] and then the resulting *dominant alarms* are repositioned using ORT (Section 3.6.1.2). Note that this pilot study differs from the evaluation of ORT (Section 3.6) in the number of verification properties and thus the number of alarms considered: *dereference of a null pointer* was considered additionally. The inclusion of alarms of this property was to eliminate any bias that can possibly get introduced to alarms of

Table 4.1: Percentage of similar alarms among the alarms resulting after applying ORT.

Application	Total Alarms	% Similar Alarms	% Same Data Dependencies	% Different Data Dependencies
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**

The column *% Same Data Dependencies* (resp. *% Different Data Dependencies*) presents percentage of the similar alarms having same (resp. different) data dependencies.

particular patterns/types, because the other alarms were analyzed in our prior study to understand reasons for stopping upward repositioning of alarms ((Section 3.6.3). Recall that the study of alarms resulting after ORT (Section 3.6.3) aimed at understanding the reasons for stopping the upward repositioning of alarms, whereas this pilot study is to understand the percentage of similar alarms that are candidates for their merging through NCD-based repositioning.

We first identified groups of similar alarms from 64,779 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 4.1. The column *Total Alarms* shows the number of total alarms generated by ORT for the selected five properties. The column *% Similar Alarms* presents percentage of similar alarms in the total alarms. The column *% Same Data Dependencies* (resp. *% Different Data Dependencies*) presents percentage of the similar alarms that have *same data dependencies* (resp. *different data dependencies*).

The study indicates that, on average, 50.89% of the alarms resulting after ORT 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.

4.4 NCDs of Similar Alarms

In this section, first we describe the notion of non-impacting control dependencies (NCDs) of alarms. Then, we discuss our approach to approximately compute NCDs of similar alarms. Lastly, we describe performing NCD-based repositioning of similar alarms by taking into account the effect of NCDs computed for them.

4.4.1 The Notion of NCD of an Alarm

Definition 4.4.1 (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 P by replacing the condition of the branching node n with a call to non-deterministic choice function[3]. We say that the dependency $n \rightarrow n'$ is an *impacting control dependency* (ICD) of ϕ only if ϕ is a false positive in P but an error in P'. Otherwise, we say that the dependency $n \rightarrow n'$ is a *non-impacting control dependency* (NCD) of ϕ. ■

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

Class 1 The variables in *cond*(ϕ_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 call to non-deterministic choice function does not cause ϕ_p to be an error.

Class 2 The variables in *cond*(ϕ_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 call to 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 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 4.2, the control dependency $n_{10} \rightarrow n_{11}$ is an ICD of A_{11}, because (i) A_{11} is a false positive as the control dependency prevents the flow of unsafe values ($i < 0$ and $i > 9$) from flowing at the alarm's program point; and (ii) the same alarm is an error[4] if the condition of the control dependency is replaced by a call to non-deterministic choice function.

$Class$ 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 call to 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 4.2, the control dependency $n_{12} \rightarrow n_{13}$ of D_{14} is NCD.

[3] A function is called non-deterministic choice function if it non-deterministically returns a value in the range of its return data type [34, 46, 153].

[4] In Figure 4.2, the call at line 4, *readInt*(), is assumed to return any of the possible values.

4.4.2 Computation of NCDs of Similar Alarms

Computing whether a given dependency e of an alarm ϕ is an ICD or NCD includes determining that ϕ is a false positive when the effect of e is captured and the same alarm is an error when the effect of e is ignored. As determining whether ϕ is a false positive is undecidable in general [54, 139], 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_{13} and A_{18} in Figure 4.2 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 4.1, 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 NCDs of similar alarms computed above approximate NCDs as defined in Definition 4.4.1. The idea of the approximation is based on the earlier observation by Kumar et al. [116] 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 there 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 4.1, if the condition $strcmp(pos, \text{“}DOWN\text{”}) == 0$ is a safety condition for D_{38}, the same condition should also have been present 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 4.1 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 number of alarms.

4.4.3 NCD-based Repositioning of Similar Alarms

To overcome the limitation of ORT, discussed in Section 4.1.2, we reposition a group of similar alarms by considering the effect of their NCDs. We design NCD-based repositioning to satisfy the following constraints $C1$, $C2$, and $C3$, 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.

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

$C3$: The number of the repositioned alarms R is 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 (i.e., r_p is a spurious error). The spurious error detection occurs only when every path between r_p and each $\phi \in \Phi'$ has an ICD of ϕ. In alarms repositioning approach, for a repositioned alarm, its corresponding alarms need to be manually inspected only when the repositioned alarm is found to be an error (Section 3.2.5). Therefore, when a repositioned alarm is a spurious error, the required inspection of its corresponding original alarms does not detect an error. As a result, in such cases, the number of alarms inspected is greater than the number of the original alarms.

To overcome the problem above—a repositioned alarm detecting a spurious error—we add the second constraint $C2$. The constraint $C2$ ensures that when a repositioned alarm is an error, at least one of its corresponding original alarms is also an error. 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 4.1). 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 4.1 also shows NCD-based repositioning of the similar alarms, obtained using the NCDs computed above (Section 4.4.2). The repositioned alarms are shown using circles. The shown NCD-based repositioning satisfies the three repositioning constraints ($C1$, $C2$, and $C3$).

During 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 $C1$).

The approximate computation of NCDs may result in identifying ICDs of a group of similar alarms as NCDs. This case arises if two or more similar alarms have different ICDs (recall that we compute transitive control dependencies of similar alarms as NCDs if their conditions are not condition-wise equivalent). In this case, a repositioned alarm, obtained based on incorrectly identified NCDs, 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 a repositioned alarm is found to be an error. We experimentally evaluate the spurious error detection rate incurred due to computing the NCDs approximately. When the approximate ICDs/NCDs computation identifies NCDs of a group of similar alarms as ICDs, NCD-based repositioning fails to reposition those alarms.

4.5 NCD-based Repositioning Technique: Algorithm

This section presents a technique for NCD-based repositioning of alarms. The technique computes ICDs of 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 ICDs separately before the repositioning is performed. In our technique, we accept all the tool-generated (original) alarms as input, and do not explicitly identify groups of similar alarms prior to their repositioning. We begin describing the technique by defining *live alarm conditions* similar to *live variables* [100].

Definition 4.5.1 (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 4.1b, condition $ny \neq 0$ is live at *exit*(n_{34}) and *entry*(n_{34}) due to the alarms D_{42} or D_{48}. However, the same condition is not live at *entry*(n_{33}).

4.5.1 Live Alarm-conditions Analysis

4.5.1.1 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*). The aim of this analysis, that we call *liveConds analysis*, is to perform NCD-based repositioning of similar alarms in Φ (Section 4.4.3). 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 (discussed later (Section 4.5.1.3).

3. The transitive control dependencies of the reposLocations of ℓ such that (i) each of the dependencies appears on a path from p to at least one reposLocations, and (ii) for every dependency, there exists a condition-wise equivalent dependency on all the paths from p to each of the reposLocations. 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, 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.

Recall the computation of avconds with their corresponding rel-alarms (Section 3.5.2) and repositioning locations and conditions (Section 3.5.3). The computation of liveConds and their associated values—values computed for liveConds—is similar to avconds and their associated values, with difference in the direction in which they are propagated during their computation. Note that, the values computed for liveConds and their relOrigAlarms form a partially ordered set. However, the set of program points that can be computed for a liveCond as its reposLocations (similar to the repositioning locations computed for an avcond) do not form a partially ordered set. Moreover, the values computed for relatedICDs of liveConds do not form a partially ordered set. Therefore, we do not call this analysis a data flow analysis (although it is very much similar to a data flow analysis), instead call it an *algorithm*. Similar to the algorithm designed for avconds analysis (Section 3.5.3.2), this liveConds analysis/algorithm still serves the purpose of computing liveConds with the intended values (discussed next). Unlike rel-alarms and avconds analyses which separately compute the rel-alarms and repositioning locations of avconds, in liveconds analysis, we compute the reposLocations and relOrigAlarms of liveconds together.

4.5.1.2 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[5] 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 $E \in 2^{\mathcal{E}_c}$. We denote 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 the following functions.

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

4.5.1.3 Performing LiveConds Analysis

LiveConds analysis computes subsets of $\mathcal{L}_b$ flow-sensitively at every program point $p \in \mathcal{P}$. Let $\mathcal{B}$ be the powerset of $\mathcal{L}_b$. We use ${}^{n}\sqcap_{\mathcal{B}}$ to denote the *merging* of the values flowing in at the *exit* of a branching node n. This merge operation is similar to a meet operation performed in data flow

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

analysis. For simplicity of discussion, below we first describe the merging of values, and then describe the computation of values at all the program points by the liveConds analysis.

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

$$S \ {}^{n}\sqcap_{\mathcal{B}} \ S' = \bigcup_{\ell \in \left(\mathit{condsIn}(S) \ \cup \mathit{condsIn}\,(S') \right)} \{\ \mathit{mergeInfo}(\ell, n, S, S')\ \} \tag{4.1}$$

$$\mathit{mergeInfo}(\ell, n, S, S') = \begin{cases} \mathit{merge}(\ell, n, A, E, A', E') & \langle \ell, A, E\rangle \in S,\ \langle \ell, A', E'\rangle \in S' \\ \langle \ell, A, E\rangle & \langle \ell, A, E\rangle \in S,\ \ell \notin \mathit{condsIn}(S') \\ \langle \ell, A', E'\rangle & \langle \ell, A', E'\rangle \in S',\ \ell \notin \mathit{condsIn}(S) \end{cases} \tag{4.2}$$

$$\mathit{merge}(\ell, n, A, E, A', E') = \mathit{meetInfo}(\ell, n, A, A',\ \mathit{meetICDsInfo}(E, E')) \tag{4.3}$$

$$\mathit{meetICDsInfo}(E, E') = \left\{\ e, e' \ \middle|\ e \in E,\ \ e' \in E',\ \text{conditions of } e \text{ and } e' \text{ are equivalent} \right\} \tag{4.4}$$

$$\mathit{meetInfo}(\ell, n, A, A', E) = \begin{cases} \langle \ell, \mathit{createReposAlarm}\,(n, A, A')\,, \emptyset\rangle & \mathit{points}(A) \neq \mathit{points}(A'), \\ & E = \emptyset \\ \langle \ell, A \cup A', E\rangle & \text{otherwise} \end{cases} \tag{4.5}$$

$$\mathit{createReposAlarm}(n, A, A') = \left\{\ \langle \mathit{entry}(n),\ \mathit{origAlarms}(A) \cup \mathit{origAlarms}(A')\rangle \right\} \tag{4.6}$$

The updates performed for the values associated with a liveCond ℓ, during the merging of values at a merge (meet) point, are described below.

1. When ℓ flows-in at the meet point through only one branch, its reposLocations and relatedICDs remain unchanged (Equation 4.2).

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 branches are different; and (iii) the relatedICDs of ℓ flowing in through both the branches do not have a condition-wise equivalent dependency (Equations 4.2 and 4.5). 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 relOrigAlarms, and the relatedICDs are updated to the control dependencies that are condition-wise equivalent (Equations 4.5 and 4.4).

To perform the liveConds analysis, we initialize the values at every program point with $\emptyset$. Then we apply the equations shown in Figure 4.3, until the fixed point is computed (similar to the maximum fixed point solution computed by a data flow analysis). These equations compute liveConds in an intraprocedural setting, along with their relOrigAlarms, reposLocations, and relatedICDs. Out_n and In_n denote the values computed by the liveConds analysis, respectively, at the *exit* and *entry* of a node n (Equations 4.7 and 4.9, respectively).

Equation 4.14 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

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

$$\mathit{Out}_n = \begin{cases} \emptyset & n \text{ is } \mathit{End} \text{ node} \\ \prod\limits_{m \in pred(n)}^{n}{}_{\mathcal{B}} \ \mathit{Edge}_{e \equiv n \to m}\,(\mathit{In}_m) & \text{otherwise} \end{cases} \tag{4.7}$$

$$\mathit{Edge}_{e\,=n\to m}(S) = \{\langle \ell, A, E \cup \mathit{handleCtrlDep}\,(e, A)\rangle \mid \langle \ell, A, E\rangle \in S\} \tag{4.8}$$

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

$$\mathit{In}_n = \mathit{Gen}_n(\mathit{Survived}_n) \cup (\mathit{Survived}_n \setminus \mathit{GenRemoved}(\mathit{Survived}_n)) \tag{4.9}$$

$$\mathit{Survived}_n = \mathit{processForICDsKill}\,(n, \mathit{Out}_n \setminus \mathit{Kill}_n\,(\mathit{Out}_n)) \tag{4.10}$$

$$\mathit{Kill}_n(S) = \left\{ \langle \ell, A, E\rangle \;\middle|\; \begin{array}{c} \langle \ell, A, E\rangle \in S, n \text{ contains a definition} \\ \text{of an operand of } \ell \end{array} \right\} \tag{4.11}$$

$$\mathit{processForICDsKill}(n, S) = \{\langle \ell, A,\ E \setminus \mathit{killICDs}(E, n)\rangle \mid \langle \ell, A, E\rangle \in S\} \tag{4.12}$$

$$\mathit{killICDs}\,(E, n) = \left\{ e \;\middle|\; \begin{array}{c} e \in E, \text{ and } n \text{ contains a definition} \\ \text{of an operand of } \mathit{cond}(e) \end{array} \right\} \tag{4.13}$$

$$\mathit{Gen}_n(S) = \left\{ \mathit{createLiveCond}(\phi, n, S) \;\middle|\; \begin{array}{c} n \text{ has alarm } \phi \in \Phi \\ \text{reported for it} \end{array} \right\} \tag{4.14}$$

$$\mathit{createLiveCond}(\phi, n, S) = \begin{cases} \mathit{createInfo}\,(\phi, n, \{\phi\} \cup \mathit{origAlarms}(R)) & \langle \ell, R, C\rangle \in S, \\ & \mathit{cond}(\phi) = \ell \\ \mathit{createInfo}\,(\phi, n, \{\phi\}) & \text{otherwise} \end{cases} \tag{4.15}$$

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

$$\mathit{GenRemoved}_n(S) = \left\{ \langle \ell, A, E\rangle \;\middle|\; \begin{array}{c} n \text{ has alarm } \phi \in \Phi \text{ reported for it,} \\ \langle \ell, A, E\rangle \in S,\ \ell = \mathit{cond}(\phi) \end{array} \right\} \tag{4.16}$$

Figure 4.3: Data flow equations of liveConds analysis.

liveCond flowing in are also added to relOrigAlarms of the reposLocation *entry*(n) (Equation 4.15); and (ii) propagation of the values of l flowing in at *entry*(n) is stopped (Equation 4.16). With this computation and the merge operation (Equation 4.1), 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 4.14) or the merge operation is performed (Equation 4.1).

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 4.8). (ii) For a relatedICD e of ℓ, if an assignment node assigns values to a variable in *cond*(ϕ), then e is removed from the relatedICDs of ℓ (Equation 4.12).

For example, in Figure 4.1b, $nx \neq 0$ and $ny \neq 0$ are two liveConds computed by the liveConds analysis at entry(n_{34}), i.e. in In_{34}. At this program point, the reposLocations (with their relOrigAlarms) and relatedICDs of the first liveCond, $nx \neq 0$, respectively are $\{\langle \mathit{entry}(n_{37}), \{D_{38}, D_{45}\}\rangle\}$ and $\emptyset$. Moreover, the reposLocations (with their relOrigAlarms) and relatedICDs of the second liveCond, $ny \neq 0$, respectively are $\{\langle \mathit{entry}(n_{41}), \{D_{42}, D_{48}\}\rangle\}$ and $\emptyset$.

Equations 4.14 and 4.11 respectively compute liveConds to be generated and killed at a node, based on liveConds that flow in at the exit of the node. Moreover, Equation 4.12 denotes that computation of ICDs associated with liveConds that flow in at the exit of the node. Hence, these equations indicate that the computation of liveConds and their associated relatedICDs by these equations is non-constant. The computation of reposLocations of liveConds is on similar lines to the computation of repositioning locations of avconds: although the reposLocations are computed as a set (of program points), their merging at a meet point is either union or same as the merging of repositioning locations of an avcond (Equation 3.25). Moreover, the updating of relatedICDs of a liveCond during the merge operation (Equation 3.25) denotes that either they remain same or get updated to ICDs having equivalent conditions. The relatedICDs computed for any liveCond at a program point are finite and unique. Therefore, the liveConds and their associated values computed by liveConds analysis at any program point converge over multiple iterations, i.e., the fixed point gets computed. Therefore, the analysis/algorithm terminates.

4.5.2 NCD-based Repositioning using LiveConds Analysis Results

Algorithm 3 presents steps to perform NCD-based repositioning using results of the liveConds analysis described above (Section 4.5.1). These steps are similar to the steps of algorithm for ORT (Algorithm 1 in Section 3.5.4). The main difference is in the first step: the first step of Algorithm 3 differs from Step 1 of ORT in terms of the identification of liveConds to compute repositioned alarms. The steps are described below.

4.5.2.1 Step 1 (Computation of Repositioned Alarms)

As discussed in Section 4.5.1, results of liveConds analysis are used to create repositioned alarms for the original alarms Φ: repositioned alarms are the alarms resulting after 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 4.4.3). Note that, the Equations 4.1, 4.8, and 4.12 together indicate that a repositioned alarm is created only when the constraint $C2$ is satisfied (Section 4.4.3).

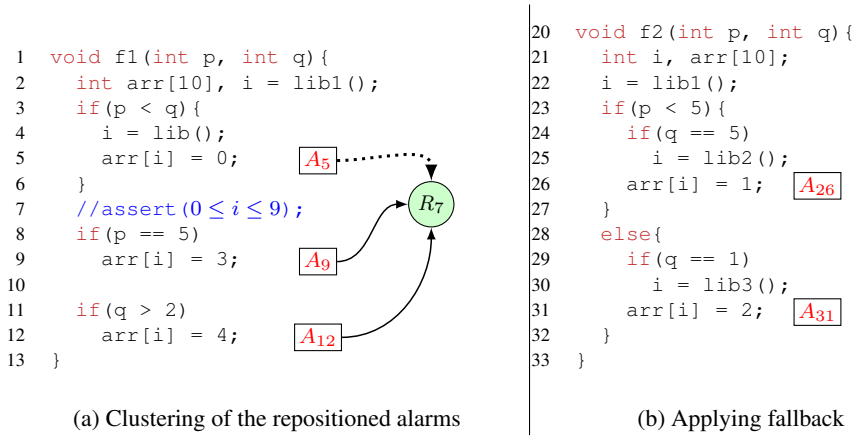
The next steps are same as the steps in the algorithm for ORT (Algorithm 1).

Algorithm 3 Steps to perform NCD-based repositioning of alarms.

```
global R_C;
procedure PERFORMNCDREPOSITIONING
    R_C = ∅;
    /* Step 1 - computation of repositioned alarms (Section 4.5.2.1) */
    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

    /* Step 2 - simplification of the repositioned alarms (Section 4.5.2.2) */
    R_S ← ∅;
    for each point p ∈ 𝒫 do
        C ← {⟨c, p⟩ | ⟨c, p⟩ ∈ R_C};
        R_S ← R_S ∪ simplifyConds(C);
    end for

    /* Step 3 - clustering of the repositioned conditions (Section 4.5.2.3) */
    R_E ← discardFollowers(R_S);

    /* Step 4 - postprocessing for fallback (Section 4.5.2.4) */
    R_f ← performFallback(R_E, Φ);   /* Algorithm 2 */

    return R_f;      /* the final repositioned alarms */
end procedure

procedure CREATEREPOSALARMS(ℓ, S)
    for each ⟨ℓ, A, E⟩ ∈ S do
        for each ⟨q, Φ'⟩ ∈ A do
            R_C = R_C ∪ {⟨ℓ, q⟩};   /* Creation of new repositioned alarms */
            for each φ ∈ Φ' do
                createLink(r, φ);  /* Add a traceability link from r to φ */
            end for
        end for
    end for
end procedure
```

4.5.2.2 Step 2 (Simplification of Repositioned Alarms)

Let R_C be the set of repositioned alarms resulting after Step 1. In this step, every program point is processed to simplify the repositioned alarms $R' \subseteq R_C$ at that point. The simplification is performed on conjunction of the conditions of repositioned alarms that are at the same point



(a) Clustering of the repositioned alarms

(b) Applying fallback

Figure 4.4: Examples to illustrate postprocessing of the repositioned alarms.

and their conditions involve checking values of the same expression. The traceability links for a condition resulting after the simplification are obtained by merging traceability links of the conditions that got simplified.

In Algorithm 3, we assume that the function *simplifyConds(C)* accepts repositioned alarms C to be simplified and returns the conditions after their simplification. Moreover, we assume that it accordingly updates traceability links of the conditions.

4.5.2.3 Step 3 (Clustering of Repositioned Alarms)

Let R_S be the set of all repositioned alarms resulting after the simplification step (Step 2). As a repositioned alarm can be a *dominant alarm* for another repositioned alarm, we postprocess R_S for their clustering using the clustering techniques [123, 150, 223]. As an example, consider the code in Figure 4.4a that has three AIOB alarms reported at lines 5, 9, and 12. Corresponding to these original alarms, the repositioned alarms resulting after the simplification step are, $R_S = \{\langle 0 \leq i \leq 9, \text{entry}(n_8)\rangle,\ \langle 0 \leq i \leq 9, \text{entry}(n_5)\rangle\}$.

Observe that the second repositioned alarm in R_S is a follower in presence of the first one (shown as an assertion on line 7). Therefore, to further reduce the number of alarms, (1) we postprocess the repositioned alarms R_S by applying the clustering techniques [124, 150, 223], and (2) discard the repositioned alarms that are identified as *followers*. Applying this step to the two example repositioned alarms, discards the follower alarm, and reduces the number of alarms in R_S by one.

When a repositioned condition is identified as a follower and discarded, its traceability links are transferred to its dominant alarm(s). In Algorithm 3, we assume that the function *discardFollowers(R_S)* performs clustering of given set of alarms R_S by implementing a clustering technique [124, 150, 223]. Moreover, we assume that the function returns only the dominant alarms, and transfers traceability links of the follower alarms to their respective dominant alarms. Let R_E be the set of repositioned alarms resulting after this clustering step.

4.5.2.4 Step 4 (Postprocessing for Fallback)

As a limitation of our technique similarly to ORT, in rare cases, repositioning of a given set of original alarms can result into more repositioned alarms than the original alarms. We illustrate this using the two AIOB alarms A_{26} and A_{31} shown in Figure 4.4b. The repositioning of these two alarms after Step 1 (Section 4.5.2.1) results in the three repositioned alarms, $R_C = \{\langle 0 \leq i \leq 9, \text{entry}(n_{26})\rangle, \langle 0 \leq i \leq 9, \text{entry}(n_{31})\rangle, \langle 0 \leq i \leq 9, \text{entry}(n_{23})\rangle\}$. Below we describe the liveConds based on which those repositioned alarms are created.

1. The repositioned alarm $\langle 0 \leq i \leq 9, \text{entry}(n_{26})\rangle$ gets created because $0 \leq i \leq 9$ is liveCond at exit(n_{25}) with entry(n_{26}) as its single reposLocation, and the same condition is not liveCond at entry(n_{25}).

2. The repositioned alarm $\langle 0 \leq i \leq 9, \text{entry}(n_{31})\rangle$ gets created because $0 \leq i \leq 9$ is liveCond at exit(n_{30}) with entry(n_{31}) as its single reposLocation, and the same condition is not liveCond at entry(n_{30}).

3. The repositioned alarm $\langle 0 \leq i \leq 9, \text{entry}(n_{23})\rangle$ gets created because $0 \leq i \leq 9$ is liveCond at exit(n_{22}) with entry(n_{23}) as its single reposLocation, and the same condition is not liveCond at entry(n_{22}). The reposLocation, entry(n_{23}), is created during the meet (merge) performed at exit(n_{23}), as the liveCond $0 \leq i \leq 9$ flowing-in at the meet point from the two branches has different reposLocations (i.e., entry(n_{26}) and entry(n_{31})).

Processing the three repositioned alarms, R_C, using Steps 2 and 3 for their simplification and clustering does not reduce their number, i.e., $R_C = R_S = R_E$. In this example, the repositioning increases the number of alarms by one. Therefore, we identify such 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 the corresponding repositioned alarms. For the example discussed above, finally A_{26} and A_{31} get reported instead of the three repositioned alarms. The algorithm to perform fallback in this step is the same as the fallback algorithm presented for ORT (Algorithm 2).

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 merging a few similar alarms, e.g., the similar alarms A_5, A_9, and A_{12} discussed above (Section 4.5.2.3). Thus, as we intend to reposition more similar alarms together, we accept all tool-generated alarms as input to NCD-based repositioning and resort to fallback in such limitation cases. That is, in this fallback step, when NCD-based repositioning of a group of similar alarms results in *equal or higher number* of repositioned alarms, we report the original alarms instead of the repositioned ones. Note that in this step, unlike in ORT fallback step, we also perform fallback when the repositioning of a group of similar alarms results in *equal number* of the repositioned alarms. We perform fallback in such cases, because we prefer to report the repositioned alarms closer to their corresponding original alarms (Section 4.4.3), whereas in ORT we reposition the alarms closer to their cause points.

Applying this fallback step ensures that the repositioning obtained using the technique satisfies the constraint $C3$ (Section 4.4.3). Thus, our technique never increases the number of alarms reported to the user than the input original alarms.

4.5.3 Properties of the NCD-based Repositioning Technique

In this section we prove properties of Algorithm 3.

Theorem 4.5.1. *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 original alarm. When the condition of the alarm, $cond(\phi_p)$ is generated as a liveCond at p, its reposLocation is p (Equation 4.14). This reposLocation 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 (Equation 4.1). Thus, the reposLocations of the liveCond $cond(\phi_p)$ at a program point q includes p or the meet points that together dominate p (because $cond(\phi_p)$ 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 alarms: a repositioned alarm is created with $cond(\phi_p)$ as its alarm condition at each of its reposLocations. Collecting the liveConds for alarms repositioning this way (i.e., by Step 1) 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 reposLocations. This is sufficient to guarantee that for every original 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 Step 1 is safe: detection of an error is not missed. Postprocessing of the repositioned alarms using Steps 2, 3, and 4 does not affect detection of an error by the repositioned alarms. Therefore, the final repositioning obtained by the NCD-based repositioning technique is safe. □

Theorem 4.5.2. *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 4.5.1.
Constraint $C2$: Equations 4.1, 4.8, and 4.12 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 alarms. A new repositioned alarm is created at this location q or at the program points of the original alarms. 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. □

4.6 Empirical Evaluation

In this section we evaluate the NCD-based repositioning technique (Algorithm 3) in terms of the reduction in the number of alarms. The evaluation is performed using alarms generated by TCS ECA on total 32 open source and industry applications.

4.6.1 Experimental Setup

Implementation We implemented the NCD-based repositioning technique (Algorithm 3) using the analysis framework of our commercial static analysis tool, TCS ECA [197]. The analysis framework supports analysis of C and COBOL programs. The framework allows to implement data flow analyses using function summaries. We implemented a liveConds analysis to compute

liveConds inter-procedurally and by considering transitivity. In the inter-procedural 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 the NCD-based repositioning technique in different contexts, we select in total 32 applications that belong to the following three categories: (i) 16 open source applications written in C and previously used as benchmarks for evaluating ORT (Section 3.6.1.2); (ii) 11 industry C applications from the automotive domain, of which four were previously used as benchmarks to evaluate ORT (Section 3.6.1.2); and (iii) 5 industry COBOL applications from the banking domain.

Compared to the applications selected during the pilot study in Section 4.3 (resp. the evaluation of ORT in Section 3.6), the above selection includes 16 (resp. 12) additional industry applications. The inclusion of additional applications was to evaluate NCD-based repositioning on applications that were not analyzed/studied previously, and thus to avoid any bias getting introduced to the set of applications used earlier.

We analyzed the applications using TCS ECA for five commonly checked categories of runtime 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 [124, 150] and then the resulting *dominant alarms* are postprocessed for their repositioning using ORT. The resulting repositioned alarms are provided as input to the NCD-based repositioning technique. All the applications in the three sets were analyzed and the alarms were postprocessed—clustering, repositioning using ORT, and the NCD-based repositioning—using a machine with i7 2.5GHz processor and 16GB RAM.

4.6.2 Evaluation Results

Table 4.2 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 the NCD-based repositioning technique, while the column *% Reduction* presents the percentage reduction achieved in the number of alarms by the technique. The evaluation results indicate that, compared to ORT, the 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 4.2 presents the time needed to (i) analyze the applications for those five properties, and (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 number of alarms may result in performance gain when the alarms are postprocessed for automated elimination of false positives using time-expensive techniques like model checking (discussed in Section 5.4). The reduction in the number of alarms reduces the number of assertions generated corresponding to the alarms, and therefore the overall number of model checking calls to be made. The reduction in the number of model checking calls reduces time taken for the automated false positives elimination.

Table 4.2: 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 applications (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

Following we describe a few other observations that we made during the evaluation.

- We separately measured 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).
- The fallback (Section 4.5.2.4) got applied in 2592 instances during the NCD-based repositioning of the total 105,546 alarms.
- We measured percentage of similar alarms among the alarms resulting from applying the NCD-based repositioning technique to the selected open source applications. Results of this study are shown under *After NCD-based repositioning* in Table 4.3 (right side). For comparison purpose, we also show the percentage of similar alarms among alarms resulting from the application of clustering and ORT, i.e., before NCD-based repositioning. The results shown on the left side are the same as those discussed in the pilot study (Section 4.3). 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, and 64% of these similar alarms appear in the repositioning limitation scenarios. Our manual analysis of 200 alarms appearing in these limitation scenarios showed that they are not merged due to (i) presence of common safety conditions (ICDs), (ii) limitations in our implementation to compute the liveConds inter-procedurally, or (iii) the fallback got applied.

Table 4.3: Percentage of similar alarms among the alarms resulting after applying ORT (left side) and alarms resulting after applying NCD-based repositioning (right side).

Application	After applying ORT				After applying NCD-based repositioning			
	Total Alarms	% Similar Alarms	% Same Data Depen-dencies	% Different Data Depen-dencies	Total Alarms	% Similar Alarms	% Same Data Depen-dencies	% Different Data Depen-dencies
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**

1. The column *Total Alarms* shows the total number of alarms generated by the techniques for the selected five properties.
2. The column *% Similar Alarms* presents percentage of similar alarms in the total alarms.
3. The column *% Same Data Dependencies* (resp. *% Different Data Dependencies*) presents percentage of the similar alarms that have *same data dependencies* (resp. *different data dependencies*).

4.6.3 Evaluation of Spurious Error Detection by Repositioned Alarms

As discussed in Section 4.4.3, 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 were randomly selected from the repositioned alarms generated on the first nine open source applications (Table 4.2a) and two industry applications (C applications 4 and 7 in Table 4.2b). These selected 150 repositioned 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, the NCD-based repositioning technique reduced the number of alarms by 70% but at the cost of detecting a few spurious errors (2%).

4.7 Related Work

As NCD-based repositioning presented in this chapter overcomes the limitation of ORT, we compared and evaluated our NCD-based repositioning technique against it.

On similar lines to alarms repositioning, Cousot et al. [39] have proposed hoisting necessary preconditions for providing the preconditions required by the Design by Contract [145]. Furthermore, Muske et al. [154] have proposed grouping related/similar alarms based on similarity of modification points. In their approach [150], 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 [39, 150, 154] identify conditional statements (control dependencies) that are non-impacting to the similar alarms.

Kumar et al. [116] identify conditional statements that are value-impacting to alarms. However, the notion of *value-impacting conditional statements* (resp. non value impacting conditional statements) is different from the ICDs (resp. NCDs) of 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 4.2, the control dependency $n_{10} \rightarrow n_{11}$ of A_{11} is ICD, whereas the technique by Kumar et al. [116] identifies the same dependency is non-value impacting. Preserving such ICD(s) on the value slice generated for an alarm can help to verify/prove that the assertion generated for the alarm holds. Otherwise, verification of the same assertion results in a counter-example. 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 leveraged 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 these combinations will provide more benefits as compared to the benefits obtained by applying them individually.

4.8 Conclusion

In this chapter, we addressed the limitation of the *original repositioning technique* (ORT). We observed that the conservative assumption about controlling conditions of alarms leads to the limited reduction in number of alarms by ORT. Based on this observation, we have proposed the notion of NCDs of alarms, and NCD-based repositioning to further reduce the number of alarms.

> Computing *non-impacting control dependencies* of alarms and taking into account their effect during repositioning of the alarms helps to improve the reduction obtained by the repositioning.

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 ORT, 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.

Compared to ORT, NCD-based repositioning reduces the number of alarms by up to 36.09% and with median reduction of 10.48%.

Our approach to compute approximated NCDs of similar alarms is based on the observation that not all controlling conditions of an alarm are actually impacting to the alarm. In the approach, a controlling condition of an alarm in a set of similar alarms is identified as ICD/NCD of the alarm depending on whether each of the similar alarms has a condition-wise equivalent controlling condition. This approximation approach is required, because a high percentage of alarms resulting after ORT are still similar, and therefore reducing their number is important.

The existing alarms clustering and repositioning techniques, being conservative, still report high percentage of similar alarms. During the evaluation of the technique, our manual analysis showed that the approach to approximately compute NCDs of similar alarms helped to reduce the number of alarms by 70%. However, the approach resulted in 2% of the repositioned alarms being spurious errors. Therefore,

Our approach to approximately compute NCDs of similar alarms is effective: the approximation helps to *safely* reduce the number of alarms while it detects only a few repositioned alarms as spurious errors.

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 explore a few more techniques to (precisely) compute NCDs for alarms (similar as well as dissimilar alarms). Precise computation of NCDs of alarms, compared to the approximation approach, can help to reduce more number of alarms and that too without detection of spurious errors by the repositioned alarms.

Chapter 5

Postprocessing of Alarms Generated on Partitioned Code

Static analysis tools used to detect common programming errors are known to generate a large number of alarms. Moreover, these tools often fail to analyze large systems. Partitioning*, splitting a large system into multiple smaller parts (called* partitions*), is commonly used to scale up these tools. Due to the conservative approach taken during analysis of the partitions, the number of alarms generated by static analysis tools on partitioned-code increases further. We find that, (1) around 45% alarms generated on partitioned-code are generated for POIs that are common to multiple partitions (called* common-POI alarms*), and (2) postprocessing these common-POI alarms partition-wise, including manual inspection, incurs redundancy. Moreover, in our study of the alarms postprocessing techniques (Section 2.6), we find that none of the alarms postprocessing techniques considers the nature of common-POI alarms generated on partitioned-code.*

To reduce the redundancy in postprocessing of common-POI alarms, we present a technique that takes into account that common-POI alarms are generated for the same POI. First we group common-POI alarms together, and propose a method to inspect them based on functions (i.e., sub-routines) identified for each group. The functions identified for each group are the topmost functions *which (1) are directly or indirectly called by each of the partitions in which the grouped alarms are generated, and (2) directly or indirectly call the function containing the POI of the alarms. Inspection of a grouped alarm in the context of the identified functions guarantees the same result for the other alarms in the same group when they are inspected in the context of the same functions. Then, we postprocess common-POI alarms for automated false positives elimination (AFPE). Existing AFPE techniques help to eliminate false positives, however they are known to have poor efficiency. To address the problem of poor efficiency, we reuse AFPE results generated for common-POI alarms across partitions. The reuse of results allows to reduce the number of calls to model checker, and thus to improve efficiency.*

Our empirical evaluation indicates that (1) the proposed method to group and inspect common-POI alarms reduces manual inspection effort by 60%; and (2) the reuse of AFPE results across partitions reduces the total AFPE time by up to 56%, with median reduction of 12.15%.

5.1 Background

In this section, we describe *code partitioning* and alarms generated through analysis of partitioned-code.

Static analysis tools have shown promise in detecting code anomalies and common programming errors [12, 14, 21, 159, 206] and even certification of safety-critical systems [24, 52, 112]. However, these tools often fail to analyze a large system as a whole [59, 78]. This is because, analysis of the whole-system demands more memory and time resources than available, and in practice, satisfying this demand is not always feasible. *Partitioning* is commonly used to scale static analysis tools to very large systems [59, 99]. Scalability is achieved by splitting a large system into smaller code parts, called *partitions*. A large system usually constitutes many functionalities (designated tasks) implemented independently of or in communication with other functionalities. The system code implementing such a functionality is considered as a partition, and is denoted by the single *sub-routine* (function) that is *entry* to the code that implements the functionality. Each partition formed, being smaller and less complex than the original system, is analyzable by static analysis tools. The partitions of a system can be provided as input to the tools, i.e., the partitioning is performed manually [59], or can be automatically identified as *uncalled functions* in the code [99], or mixture of both. Note that any two partitions created, either manually or automatically, can be overlapping (resp. mutually exclusive) if the sets of functions transitively called[1] by the two partitions are not disjoint (resp. are disjoint).

In general static analysis tools are known to generate a large number of alarms [37, 92, 120]. Due to the following two reasons, alarms generated on partitioned-code are more in number than the alarms that would be generated on the corresponding non-partitioned code.

1. *Conservative Approach for Shared Variables*: During analysis of partitioned-code, each partition is analyzed separately to produce results specific to that partition. To ensure soundness of the analysis results, analysis of each partition assumes all values are possible for the variables that are also modified by other partitions (*shared variables*). The conservative approach adopted for shared variables increases the number of alarms generated on partitioned-code compared to the corresponding non-partitioned code.

2. *Presence of Common-POI Alarms*: A point-of-interest (POI) to be checked by a static analysis tool can appear in multiple partitions, because the function containing the POI can be transitively called in multiple partitions, and an alarm can be generated for the POI in two or more partitions. We refer to the alarms that are generated for the same POI but appearing in multiple partitions as *common-POI alarms*. The presence of common-POI results in generating more alarms on partitioned-code than the number of alarms that would be generated on the corresponding non-partitioned code: alarms generated on the partitioned-code is the union of the sets of alarms generated on each of the partitions.

   For example consider the partitioned-code in Figure 5.1, having two partitions *p1* and *p2*. Functions *foo* and *bar* are common to these partitions, as they are transitively called in both the partitions. Analyzing partition *p1* using a static analysis tool for *array index out of bounds* (AIOB) and *division by zero* (DZ) verification properties generates one AIOB and one DZ alarm. Henceforth, we use notation $\langle \phi, p \rangle$ to denote an alarm ϕ generated in partition p. The two alarms generated in *p1* are $\langle A_{30}, p1 \rangle$ and $\langle D_{34}, p1 \rangle$. Analysis of the other partition, *p2*, for those two properties generates four alarms: $\langle D_{14}, p2 \rangle$, $\langle A_{30}, p2 \rangle$,

[1] We call a function g is transitively called by a function f if f calls g directly or indirectly.

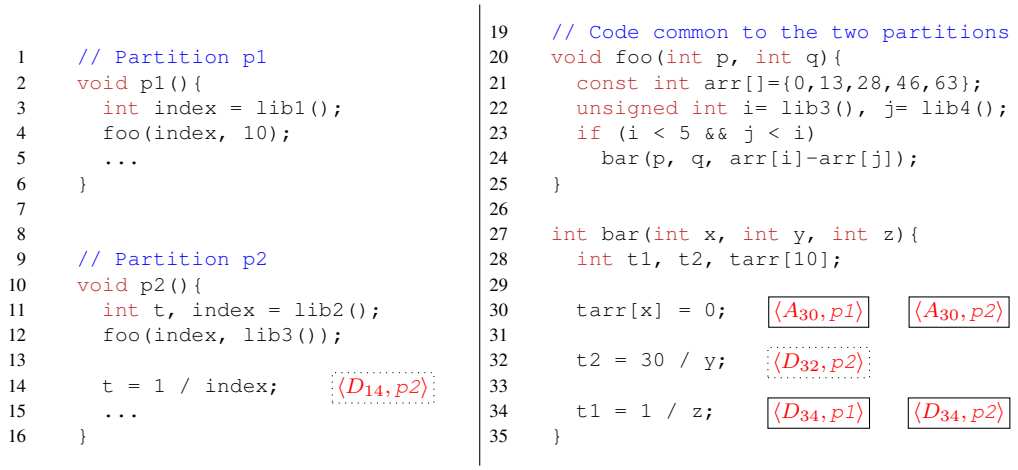


Figure 5.1: Examples of alarms generated on partitioned code. An alarm ϕ generated in partition p is shown using $\langle \phi, p \rangle$. The alarms shown in dotted rectangles are *unique alarms* while the other alarms are common-POI alarms.

$\langle D_{34}, p2 \rangle$, and $\langle D_{32}, p2 \rangle$. When the code is not partitioned, i.e., when those two functions (partitions) *p1* and *p2* are called from the same application, in addition to the two alarms generated at lines 14 and 32, only one alarm gets generated at lines 30 and 34.

We use *unique alarms* to refer to the alarms other than common-POI alarms. For example, $\langle D_{32}, p2 \rangle$ is a unique alarm: although the alarm's POI also appears in partition *p1*, an alarm is not generated in *p1* for the same POI. As another example, $\langle D_{14}, p2 \rangle$ is a unique alarm, because its POI appears only in one partition *p2*.

We use *context of a function* to refer to the code in that function and the functions transitively called by that function. The values passed to parameters of a function are assumed to be outside the context of the function. Since a partition is denoted using a function, context of a partition is same as context of the function that denotes the partition.

5.2 Motivation

In this section, we describe the problem of redundancy in postprocessing of common-POI alarms generated during analysis of partitioned-code, and present an overview of our solution proposed to address this redundancy problem.

5.2.1 The Problem

As discussed above, analysis of partitioned-code increases the number of alarms as compared to the corresponding non-partitioned code. To measure percentage of common-POI alarms in the alarms generated on partitioned-code, we performed a pilot study using alarms generated by TCS ECA [197] on two industry partitioned-code applications. Section 5.5.1 presents details of the two applications. In the study, we found that 45% of alarms generated on the partitioned-code are common-POI alarms.

Below we illustrate redundancy in postprocessing common-POI alarms using existing techniques, by referring to the alarms shown in Figure 5.1. We limit the scope of postprocessing only to common-POI alarms and the following two postprocessing approaches: *simplification of manual inspection* and *automated false positives elimination* (AFPE).

1. **Redundancy in Manual Inspection of Alarms:** All the common-POI alarms generated on partitioned code need to be inspected manually, because detection of an error by each common-POI alarm varies depending on the partition in which it is generated. For example, $\langle A_{30}, p1 \rangle$ can be an error while $\langle A_{30}, p2 \rangle$ can be a *safe* program point[2], and vice versa. Moreover, a unique alarm requires inspecting it in the context of the partition in which it is generated. Therefore, the alarms generated on partitioned code are inspected partition-wise.

   The partition-wise manual inspection of common-POI alarms results in inspecting the same code repeatedly and thus incurs redundancy. For example, inspection of $\langle D_{34}, p1 \rangle$ requires identifying values of x at line 34, and the identification requires traversing the code backward. During the inspection, the alarm gets determined as a false positive considering the code in function *bar*. The denominator in the alarm takes values computed by the expression *arr[i] - arr[j]* at line 24. Due to the subtraction of two different values in the array *arr*, the expression always computes non-zero values. Along similar lines, the other common-POI alarm, $\langle D_{34}, p2 \rangle$, also gets determined as a false positive considering the same code in function *bar*.

   Note that, the result of inspection of $\langle A_{30}, p1 \rangle$ and $\langle A_{30}, p2 \rangle$ depends on their partitions, because the values of the index variable x are taken transitively from the values returned by the calls to two different libraries at lines 3 and 11. Inspecting these two alarms separately in their partitions incurs *partial* redundancy: traversing the code backward from the alarm POI to the start of function *foo* is common in inspection of the two alarms.

2. **Redundancy in AFPE:** To improve scalability of model checker used, AFPE techniques implement context expansion [34, 153, 174] (see Section 5.4.1). We find that applying context expansion to common-POI alarms incurs redundancy. For example, postprocessing $\langle D_{34}, p1 \rangle$ using the AFPE techniques results in verifying the corresponding assertion in two verification contexts: first *bar* and then *foo*. The alarm gets eliminated after verification in context *foo*. Similarly, the same two verification calls are also made for $\langle D_{34}, p2 \rangle$. This indicates applying AFPE techniques to common-POI alarms results in making repeated verification calls, and thus the elimination incurs redundancy. Section 5.4.2 discusses the redundancy problem in detail.

In our survey of the techniques proposed for postprocessing of alarms (Section 2.6), we found that none of the postprocessing techniques specifically postprocess alarms generated on partitioned-code. To eliminate the redundancy in manual inspection and AFPE of common-POI alarms (discussed above), we ask the following research questions.

RQ 4: How can we reduce redundancy in manual inspection of common-POI alarms?

[2]We assume that, the library call *lib1()* (resp. *lib2()*) returns values such that the correspondingly generated common-POI alarm $\langle A_{30}, p1 \rangle$ (resp. $\langle A_{30}, p2 \rangle$) can be a false positive.

RQ 5: How can we reduce redundancy in AFPE applied to common-POI alarms?

5.2.2 Our Solution

To reduce the redundancy in postprocessing of common-POI alarms, we postprocess them by taking into account that they are generated for the same program point. First, aiming at reducing the redundancy in manual inspection of common-POI alarms, we group the alarms together and identify functions for each group. The functions identified for each group are the *topmost functions* which (1) are directly or indirectly called by each of the partitions in which the grouped alarms are generated, and (2) directly or indirectly call the function containing the POI of the alarms. The grouping and identification of the functions for each group is such that inspection of a grouped alarm in the context of the identified functions guarantees the same result for the other alarms in the same group when they are inspected in the context of the same functions.

Based on the groups formed and functions identified for them, we propose a method to inspect the grouped common-POI alarms. The proposed method helps to *safely* avoid redundant inspection of common-POI alarms: when a grouped alarm is identified as a false positive in the context of each of the functions identified for the group, *all the alarms* in its group are also false positives.

Then, we postprocess common-POI alarms using AFPE techniques. To reduce the redundancy in AFPE, we reuse AFPE results of the common-POI alarms across partitions. The reuse of results allows to reduce the number of calls to the model-checker, and thus to improve efficiency.

Note that, the postprocessing proposed in this chapter is applicable only to common-POI alarms. Unlike common-POI alarms, processing the unique alarms by existing postprocessing techniques does not incur redundancy. Therefore, we exclude them from our discussion.

Our evaluations performed using alarms generated by TCS ECA [197] indicated the following.

- The proposed method to group and inspect common-POI alarms reduces manual inspection effort by 60%. The reduction is considering inspection of common-POI alarms only (unique alarms were not considered in this evaluation).
- The proposed reuse of AFPE results across partitions reduces the total AFPE time by up to 56%, with median reduction of 12.15%. The total AFPE time included time required to process all the alarms using AFPE techniques: common-POI as well as unique alarms.

Following are the key contributions of our work presented in this chapter.

1. A grouping-based method to reduce the redundancy involved in manual inspection of common-POI alarms.
2. A technique to reduce the redundancy in AFPE when applied to common-POI alarms.

Chapter Outline Section 5.3 describes the proposed technique to reduce the redundancy in manual inspection of common-POI alarms. Section 5.4 presents the proposed technique to reduce the redundancy in AFPE applied to common-POI alarms. Section 5.5 describes our empirical evaluation. Section 5.6 presents related work, and Section 5.7 concludes.

5.3 Manual Inspection of Common-POI Alarms

This section describes our proposed grouping of common-POI alarms (Section 5.3.2), and an inspection method to reduce the redundancy in their manual inspection (Section 5.3.3). We begin by defining the terms used to describe the grouping and inspection method.

5.3.1 Definitions

Definition 5.3.1 (Reversed Call Chains of an Alarm)**.** For an alarm $\langle \phi, p \rangle$ with f as its immediate enclosing function, we call a list of functions a *reversed call chain of the alarm*, if

1. f is the first function in the list;
2. each function in the list is transitively called from p, and does not belong to a recursive chain, i.e., it does not call itself directly or indirectly; and
3. except *f*, every other function in the list is an immediate caller of its previous function in the list. ■

For example, in Figure 5.1, the possible *reversed call chains* of $\langle D_{34}, p1 \rangle$ and $\langle A_{30}, p1 \rangle$ are *bar*, *bar* ← *foo*, and *bar* ← *foo* ← *p1*. We use arrow in the lists to denote *calls* relationship: the function on the right of the arrow calls the function on the left. On similar lines, the possible *reversed call chains* of $\langle D_{34}, p2 \rangle$ and $\langle A_{30}, p2 \rangle$ are *bar*, *bar* ← *foo*, and *bar* ← *foo* ← *p2*.

Definition 5.3.2 (Top of a Reversed Call Chain)**.** We call the function at the end of a reversed call chain *top of the chain*. ■

For example, *f1* is the *top* of the following chains: *f1*, *f2* ← *f1*, and *f3* ← *f2* ← *f1*.

Definition 5.3.3 (Overlapped Reversed Call Chains of Common-POI Alarms)**.** We call a reversed call chain that is common to a group of common-POI alarms an *overlapped reversed call chain* (ORCC) of those common-POI alarms. In other words, for a set of common-POI alarms A, we say that a reversed call chain is an ORCC of A if the call chain is a reversed call chain of every alarm in A. ■

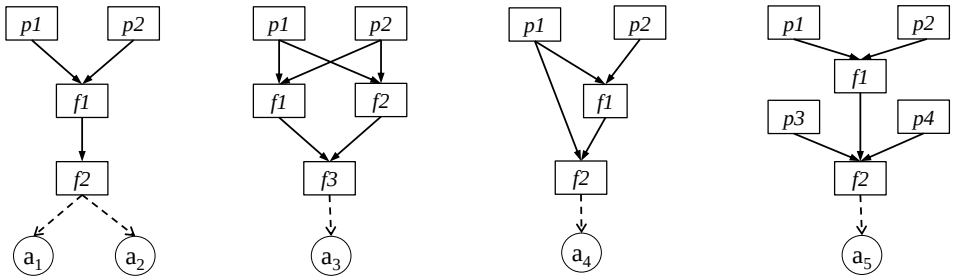
For example, in Figure 5.1, *bar* and *bar* ← *foo* are ORCCs of common-POI alarms $\langle D_{34}, p1 \rangle$ and $\langle D_{34}, p2 \rangle$.

Consider the examples of common-POI alarms in Figure 5.2, shown using circles. The rectangles denote functions; *p1*, *p2*, *p3*, and *p4* denote the partitions[3] in which the alarms are generated. The ORCCs of common-POI alarms $\langle a_1, p1 \rangle$ and $\langle a_1, p2 \rangle$ (also $\langle a_2, p1 \rangle$ and $\langle a_2, p2 \rangle$), shown in Figure 5.2(a), are *f2* and *f2* ← *f1*. The ORCCs of the common-POI alarms $\langle a_3, p1 \rangle$ and $\langle a_3, p2 \rangle$, shown in Figure 5.2(b), are *f3*, *f3* ← *f1*, and *f3* ← *f2*. The ORCCs of common-POI alarms shown in Figure 5.2(c) are *f2* and *f2* ← *f1*. There is only one ORCC, *f2*, for the four common-POI alarms shown in Figure 5.2(d).

Definition 5.3.4 (Strict-ORCCs of Common-POI Alarms)**.** Let A be a group of common-POI alarms, and P be the set of partitions in which the alarms are generated. We call an ORCC of A *strict-ORCC* if

1. every partition (i.e., function) in P transitively calls the top of the ORCC;

[3]Recall that a partition is denoted by the top most function in that partition (Section 5.1).



(a) (b) (c) (d)

Figure 5.2: Examples of common-POI alarms, shown using circles. The rectangles denote functions; *p1*, *p2*, *p3*, and *p4* denote the partitions in which the alarms are generated. The solid arrows denote *calls* relationship between two functions, while the dotted arrows denote alarms generated in the functions.

2. there exists an immediate caller f_c of the top of the ORCC such that f_c is not transitively called by at least one function in P; and

3. for each function f in the ORCC, when f is not the top of the ORCC, every immediate caller of f is transitively called by all the functions in P. ■

For example, following are the strict-ORCCs of alarms shown in Figure 5.2.

- The common-POI alarms $\langle a_1, p1 \rangle$ and $\langle a_1, p2 \rangle$ in (a) have only one strict-ORCC: *f2* $\leftarrow$ *f1*.
- *f2* is the only strict-ORCC of the common-POI alarms shown in (c) and (d).

Note that, for a given set of common-POI alarms, there can exist multiple strict-ORCCS. For example, the common-POI alarms shown in Figure 5.2(b) have two strict-ORCCs: *f3* $\leftarrow$ *f1*, and *f3* $\leftarrow$ *f2*.

5.3.1.1 Properties of Strict-ORCCs of Common-POI alarms

Let Φ be a group of common-POI alarms generated in partitions P, and the POI of the alarms is in function f_0. Recall that a partition is denoted using the function that is entry to the code represented by that partition (Section 5.1). Therefore, P is a set of at least two functions. Let O_s be the set of strict-ORCCs of Φ, and T be the set of the tops of each chain in O_s. We make the following observation for f_0 and the functions in T.

Theorem 5.3.1. *Every direct or indirect call to f_0 from every partition in P invokes exactly one function in T.*

Proof. We start by observing that no function in T calls (directly or indirectly) another function in T. This follows from the fact that, an immediate caller of a top function in T is not transitively called by at least one function in P (point 2 of definition 5.3.4), i.e., no transitive caller of the top function can be a top of any strict-ORCC in O_s. Therefore, to prove the theorem, it is sufficient to prove that every direct or indirect call from every partition in P to f_0 includes invoking a function in T. We prove this by proving it separately for direct and indirect calls from any partition $p \in P$ to f_0.

When the call from p to f_0 is a direct call, by definition of the strict-ORCCs, f_0 is the only strict-ORCC for the common-POI alarms Φ, and $T = \{f_0\}$. Therefore, a direct call to f_0 invokes the function in T.

Let $f_0 \leftarrow ... \leftarrow f_n \leftarrow p$, where $n \geq 1$, be a reversed *indirect call* from p to f_0. A prefix of this reversed call chain will always overlap with at least one of the reversed call chains for all other common-POI alarms generated for the same POI but in other partitions, i.e., the prefix is an ORCC of Φ. We denote this prefix using $\textit{prefix}_1 = f_0 \leftarrow ... \leftarrow f_k$, where $0 \leq k \leq n$. If $\textit{prefix}_1$ is a strict-ORCC of Φ, its top belongs to T, and therefore the indirect call from p to f_0 includes invoking a function in T. If $\textit{prefix}_1$ is not a strict-ORCC of Φ, there will exist a prefix $\textit{prefix}_2$ of $\textit{prefix}_1$ such that $\textit{prefix}_2$ is a strict-ORCC of Φ: f_0 is the smallest possible $\textit{prefix}_2$ that definitely exists if any other longer prefix does not exist as a strict-ORCC. Therefore, the top of $\textit{prefix}_2$ will be in T. It indicates that the indirect call from p to f_0 invokes a function in T. □

Theorem 5.3.2. *If an alarm in A gets identified as a false positive in the context of every function in T, all the alarms in A are guaranteed to be false positives.*

Proof. We first prove that when a common-POI alarm $\phi \in \Phi$ generated in a partition $p \in P$ is a false positive in the context of every function in T, ϕ is also a false positive in the context of p. Then we prove that the same is true for all the other common-POI alarms in Φ. By Theorem 5.3.1, every call from the partition $p \in P$ to f_0 includes invoking one of the functions in T. Consequently, all possible calls from each partition $p \in P$ to f_0 pass through functions in T. Therefore, when ϕ is a false positive in the context of each of the functions in T, ϕ is also a false positive in the context of p: every function in T is called directly or indirectly by p. The functions in T are computed based on the partitions in which the common-POI alarms Φ are generated, and the functions are common to the group of common-POI alarms. Hence, when a common-POI alarm in Φ is a false positive in the context of every function in T, all the other common-POI alarms in Φ are also false positives. □

5.3.2 Grouping of Common-POI Alarms

Recall that manual inspection of common-POI alarms suffers from redundancy, due to inspecting the same code repetitively (Section 5.2.1). To eliminate this redundancy, we group common-POI alarms together. For each group formed, we compute and report the *top* of each of the *strict-ORCCs* of the grouped alarms. We call the tops computed for each group *tops of the group*. The reporting of tops for each group is based on the observation described above (Section 5.3.1.1): if a grouped alarm gets adjudged as a false positive in the context of all those *top* functions, all the alarms in its group are also false positives and they are eliminated together.

Table 5.1 presents grouping of common-POI alarms in Figure 5.2, and proposed reporting of the groups. The table presents the groups (with unique IDs assigned to them), common-POI alarms in each group and the partitions in which they are generated, and *tops of each group*. For each reported top of a strict-ORCC, we also report the length of the strict-ORCC. The reported length indicates the distance of the *top* function, in terms of the number of functions, from the immediately enclosing function of the grouped alarms. The enclosing function is assumed to be at a distance of one.

We expect that, common-POI alarms in a group are more likely to be false positives in the context of tops of longer strict-ORCCs compared to tops of the shorter ones, because longer strict-ORCCs cover more code context. Therefore, we further create (sub)-groups of the common-POI alarms in each group, so that the common-POI alarms in the sub-groups can have longer

Table 5.1: Reporting of groups of common-POI alarms shown in Figure 5.2

Figure	Group ID	Sub-group ID	Common-POI alarm	Partition	Tops of the sub-groups (length)	Tops of the group (length)
Figure 5.2(a)	1	-	$\langle a_1, p1 \rangle$	*p1*	-	*f1*(2)
			$\langle a_1, p2 \rangle$	*p2*		
	2	-	$\langle a_2, p1 \rangle$	*p1*	-	*f1*(2)
			$\langle a_2, p2 \rangle$	*p2*		
Figure 5.2(b)	3	-	$\langle a_3, p1 \rangle$	*p1*	-	*f1*(2), *f2*(2)
			$\langle a_3, p2 \rangle$	*p2*		
Figure 5.2(c)	4	-	$\langle a_4, p1 \rangle$	*p1*	-	*f1*(2)
			$\langle a_4, p2 \rangle$	*p2*		
Figure 5.2(d)	5	1	$\langle a_5, p3 \rangle$	*p3*	*f2*(1)	*f2*(1)
			$\langle a_5, p4 \rangle$	*p4*		
		2	$\langle a_5, p1 \rangle$	*p1*	*f1*(2)	
			$\langle a_5, p2 \rangle$	*p2*		

strict-ORCCs compared to the strict-ORCCs of alarms in the main group. We perform sub-grouping only if (1) two or more sub-groups can be formed, each having least two alarms; and (2) common-POI alarms in at least one sub-group have longer strict-ORCCs than the strict-ORCCs of all the common-POI alarms in the group. In the reporting of alarms, for each sub-group, we also report the *top* and *length* of every strict-ORCCs computed for the alarms in the sub-group. Table 5.1 also presents sub-groups formed for alarms in Figure 5.2(d). The alarms $\langle a_5, p1 \rangle$ and $\langle a_5, p2 \rangle$ are sub-grouped together, because the strict-ORCC computed for them is longer than the strict-ORCC computed for the grouped four common-POI alarms.

Note that, when a caller of the enclosing function of an alarm belongs to a recursive chain, we do not compute reversed call chains (Definition 5.3.1). Hence, strict-ORCCs are not computed for a group of common-POI alarms if their immediate enclosing function is called from a function in a recursive chain. In such cases, as a limitation of our grouping method, common-POI alarms are not grouped together: we report them similar to unique alarms (Section 5.1).

5.3.3 Manual Inspection of Grouped Alarms

We propose manual inspection of the grouped common-POI alarms based on the *tops* computed for the groups. The inspection of grouped common-POI alarms is partition-wise, and includes the following:

a) During inspection of alarms generated for partition p, common-POI alarms generated in that partition are inspected along with the unique alarms generated for it. During the inspection, if a common-POI alarm generated for the current partition is identified as a false positive in the context of each of the tops reported for the alarm's group, all the alarms in the same group are marked as false positives and eliminated together.

b) if it is not the case in (a), and if the alarm is identified as a false positive in the context of tops of its sub-group, all the alarms in its sub-group are eliminated together.

5.4 Efficient Elimination of False Positives

This section first describes model checking-based techniques for automated false positives elimination (AFPE) and the problem of poor efficiency of those techniques. It then describes our approach to improve efficiency of the techniques by reducing redundancy in the elimination.

5.4.1 AFPE Techniques: Background

Recall that AFPE techniques eliminate false positives from alarms using more precise techniques like model checking, symbolic execution, and deductive verification (Section 2.4.4). The techniques that use model checking generate an assertion corresponding to each alarm such that the alarm is a false positive when the assertion holds, e.g., corresponding to alarm $\langle D_{34}, p2 \rangle$ shown in Figure 5.1, an assertion having condition $z \neq 0$ is created. Then they verify the generated assertion using a model checker [35, 47, 174, 205, 220]. When the assertion is found to hold, the corresponding alarm is identified as a false positive and is eliminated. However, due to the state space problem, model checking-based verification is known to suffer from non-scalability (inability to analyze large programs) and poor performance [34, 153, 174].

To scale model-checking based AFPE on large systems, Post et al. [174] have proposed a *context expansion approach*. Verification of an assertion is started from the function that includes the assertion (the lowest context). Then the verification context (function in which the assertion is verified) is expanded to its callers until the assertion is proven to hold, a verification call times-out or runs out-of-memory, or the last verified context is the entry-function of the partition. Verification of assertion(s) in the context of a function is performed under the assumption that all values are possible for variables that are input to the function and the functions called by it. As a result, when an assertion is found to hold in some function, its corresponding alarm is guaranteed to be false positive. The context expansion approach has been adopted in several works and has helped to use the model checker in a more scalable way [34, 152, 153].

We use *mcall*(ϕ, f) to denote a call to model checker (*model checking call*) which verifies the assertion generated corresponding to an alarm ϕ, and in the context of a given function f. Since the verification context in model checking calls is specified as a function, we use the terms *context* and *function* interchangeably.

Poor Efficiency of AFPE Techniques Although the context expansion adopted by AFPE techniques has helped them to use a model checker in a more scalable way, the approach considerably increases the number of model checking calls, and hence, further degrades efficiency of the techniques. For example, consider alarm $\langle D_{34}, p1 \rangle$ shown in Figure 5.1, i.e., D_{34} generated in the partition *p1*. Applying the context expansion-based AFPE technique for this alarm results in two model checking calls: *mcall*$(\langle D_{34}, p1 \rangle, bar)$ and *mcall*$(\langle D_{34}, p1 \rangle, foo)$. The alarm is identified as a false positive and eliminated based on the second model checking call. On similar lines, applying the technique for $\langle A_{30}, p1 \rangle$ results in three model checking calls: *mcall*$(\langle A_{30}, p1 \rangle, bar)$, *mcall*$(\langle A_{30}, p1 \rangle, foo)$, and *mcall*$(\langle A_{30}, p1 \rangle, p1)$. Note that the two alarms $\langle D_{34}, p1 \rangle$ and $\langle A_{30}, p1 \rangle$, generated in the same partition *p1*, are processed separately. The separate processing is intended to reduce the state space, and thus to improve scalability of model checking. These examples indicate that, multiple model checking calls are performed for each alarm, and therefore processing the tool-generated alarms using the AFPE techniques results in a very large number of model checking calls.

Grouping of assertions [34, 152] has been proposed to create groups of related assertions, and verify assertions in each group together. However, the number of generated groups of re-

lated assertions is still large, and the context expansion approach gets applied to each group. For example, the grouping techniques fail to group together and verify assertions generated for the alarms $\langle D_{34}, p1 \rangle$ and $\langle A_{30}, p1 \rangle$ shown in Figure 5.1. In addition to the context expansion approach, AFPE techniques [34, 45, 154] use *program slicing* to prune the code before each model checking call, which increases the time taken by AFPE. Evaluations of the AFPE techniques [34, 45, 152, 153] that are based on the context expansion approach, program slicing, and grouping of related assertions, indicate that processing a group of related assertions, on an average, involves making five model checking calls and takes around three to four minutes.

As a result of the large number of model checking calls made during AFPE and each call taking considerable amount of time, applying AFPE to alarms becomes time-consuming and ultimately renders AFPE unsuitable for postprocessing of alarms generated on large systems. For example, processing 200 groups of related alarms would require more than 10 hours.

5.4.2 Redundancy in AFPE Applied to Common-POI Alarms

We observe that applying context expansion-based AFPE technique to common-POI alarms results in repeated model checking calls. For example, applying the techniques to $\langle D_{34}, p1 \rangle$ shown in Figure 5.1, results in calls *mcall*($\langle D_{34}, p1 \rangle$, *bar*) and *mcall*($\langle D_{34}, p1 \rangle$,*foo*). Applying the techniques to $\langle D_{34}, p2 \rangle$ results in calls *mcall*($\langle D_{34}, p2 \rangle$, *bar*) and *mcall*($\langle D_{34}, p2 \rangle$,*foo*). Note that, the two model checking calls made for these two common-POI alarms, $\langle D_{34}, p1 \rangle$ and $\langle D_{34}, p2 \rangle$, are the same and their results will also be the same. On similar lines, among the three model checking calls made for $\langle A_{30}, p1 \rangle$ and $\langle A_{30}, p2 \rangle$, the two calls made in the contexts of *bar* and *foo* are also the same. This indicates applying AFPE techniques to common-POI alarms incurs redundancy. Observe that, this redundancy is similar to the redundancy in manual inspection of common-POI alarms (Section 5.2.1).

In Table 5.2, we illustrate this redundancy problem by showing the model checking calls made for each of the common-POI alarms shown in Figure 5.2. We denote the model checking calls by using only the *verification contexts* (functions) in those calls. The calls are shown assuming that the assertion corresponding to each common-POI alarm does not hold in any of the functions, including the function that denotes the partition of the alarm. We also assume that the context expansion is performed in *depth-first* manner. The calls shown in bold are repeated calls. The repeated calls are identified assuming that the shown alarms are processed in the order of the rows in the table.

5.4.3 Our Solution

To eliminate redundancy in processing common-POI alarms using AFPE techniques, we implement a reuse-based technique similar to *memoization* [2] and *tabling* [177, 196]. That is, before making a model checking call, we check whether the same call has been already performed for a different alarm generated for the same POI but in another partition. If the call has been already performed, we reuse result of the earlier call, otherwise the call is made. This way reusing results of model checking calls across partitions for common-POI alarms allows to reduce the number of model checking calls and thus improve efficiency.

For example, applying this reuse-based approach to alarms shown in Figure 5.2 allows to skip the repeated model checking calls in Table 5.2 (shown in bold) .

Table 5.2: Model checking calls that will be made during AFPE applied to common-POI alarms shown in Figure 5.2. The calls are shown using only the *verification contexts* (functions) in them. The contexts shown in bold denote the repeated calls.

Figure	Common-POI alarm	Partition	Model checking calls	Total calls	Repeated calls across partitions
Figure 5.2(a)	$\langle a_1, p1\rangle$	*p1*	*f2*, *f1*, *p1*	12	4
	$\langle a_1, p2\rangle$	*p2*	***f2***, ***f1***, *p2*		
	$\langle a_2, p1\rangle$	*p1*	*f2*, *f1*, *p1*		
	$\langle a_2, p2\rangle$	*p2*	***f2***, ***f1***, *p2*		
Figure 5.2(b)	$\langle a_3, p1\rangle$	*p1*	*f3*, *f1*, *p1*	6	2
	$\langle a_3, p2\rangle$	*p2*	***f3***, ***f1***, *p2*		
Figure 5.2(c)	$\langle a_4, p1\rangle$	*p1*	*f2*, *f1*, *p1*	6	2
	$\langle a_4, p2\rangle$	*p2*	***f2***, ***f1***, *p2*		
Figure 5.2(d)	$\langle a_5, p1\rangle$	*p1*	*f2*, *f1*, *p1*	10	4
	$\langle a_5, p2\rangle$	*p2*	***f2***, ***f1***, *p2*		
	$\langle a_5, p3\rangle$	*p3*	***f2***, *p3*		
	$\langle a_5, p4\rangle$	*p4*	***f2***, *p4*		

5.5 Experimental Evaluation

In this section we evaluate the grouping-based method to reduce redundancy in manual inspection of alarms, and the reuse-based technique to eliminate the redundancy in AFPE applied to common-POI alarms.

5.5.1 Evaluation of the Grouping-based Inspection Method

In the evaluation, we selected the following partitioned industry applications.

1. **App 1:** An infotainment system having 14 million lines of code and 31288 functions. The system was partitioned into 98 partitions, which run in parallel. These partitions were manually identified by designers and maintainers of the system. The size of the partitions varied from 4 to 700 KLOC. Among the total 31288 functions, 9327 were common to two or more partitions.

2. **App 2:** An embedded system, having 10 KLOC, which was partitioned into 29 partitions. These partitions were identified automatically (all uncalled functions were treated as partitions). This system spanned over 83 functions of which 14 were common to two or more partitions.

We analyzed the partitions of the two applications[4], using TCS ECA [197], for commonly checked properties *zero division* (DZ), *array index out of bound* (AIOB), *illegal dereference of a pointer* (IDP), and *overFlow-underFlow* (OFUF). We performed the proposed grouping of common-POI alarms and computed *tops* for the groups. Table 5.3 presents the results of the

[4]This evaluation was performed before the repositioning techniques are evaluated (Chapters 3 and 4). As these two applications were no longer accessible, they were not included in the evaluations of the repositioning techniques.

Table 5.3: Details of groups of common-POI alarms and their tops.

Appli-cation	Property	Total Alarms	Grouped Common-POI alarms	Total Groups	Groups with 5+ alarms	Groups with multiple tops	Groups having sub-groups	Mean length of strict-ORCCs of (sub)-groups		Time Taken (seconds)	
								Groups	Sub-groups	Group-ing	Sub-group-ing
App 1	DZ	825	494	117	53	2	53	2.23	3.11	1	2
	AIOB	12946	5765	1199	229	42	369	2.61	3.43	22	75
	IDP	99762	52412	8630	2234	338	3473	2.48	2.84	67	620
	OFUF	100066	38897	7515	1989	661	2773	2.78	3.11	41	324
App2	DZ	5	4	2	0	0	0	1.5	0	0.1	0
	AIOB	72	0	0	0	0	0	0	0	0.5	0
	IDP	559	173	48	11	11	25	1.25	1.74	1	1
	OFUF	321	53	15	3	3	10	1	1.6	1	1

Table 5.4: Results of manual inspection of the selected groups of common-POI alarms.

Appli-cation	Property	Total groups inspected	(Sub)-groups and alarms eliminated together by inspecting a single alarm from the (sub)-groups (Section 5.3.3)			
			Number of groups	Alarms in the groups	Number of sub-groups	Alarms in the sub-groups
App 1	AIOB	80	65	221	4	13
	OFUF	60	40	224	3	6
App 2	DZ	2	1	2	0	0
	IDP	23	4	9	3	7
	OFUF	15	9	31	2	4

analysis by TCS ECA and the grouping of common-POI alarms. The results indicate that around 45% of the alarms generated on those partitions are common-POI alarms. Since the groups of common-POI alarms are formed per POI, the number of groups is equal to the number of POIs for which the common-POI alarms are generated. Therefore, the number of POIs for which these common-POI alarms are generated is 18% of the number of common-POI alarms. Around 25% of the total groups formed include five or more alarms.

In order to evaluate the reduction in effort required to inspect common-POI alarms based on the proposed method, we randomly selected 180 groups of common-POI alarms. The selected groups had in total 640 alarms. We manually inspected these alarms using the method proposed to inspect grouped common-POI alarms. We measured the number of groups (resp. sub-groups) in which the grouped alarms were identified and eliminated as false positives by inspecting a single alarm from each of those groups (resp. sub-groups) (Section 5.3.3). Table 5.4 presents results of this activity. The results indicate the following:

1. Inspection of 131 alarms collectively lead to elimination of 517 alarms.

2. Around 66% (119 out of 180) groups got eliminated just by inspecting a single alarm from each group.

3. 6% of the eliminated alarms were eliminated based on the *tops* of their sub-groups. This indicates that the sub-grouping of alarms is useful to reduce redundancy in manual inspection when the high-level grouping fails to do so.

To evaluate reduction in manual inspection effort, that can be obtained by the proposed method to group and inspect common-POI alarms, the author manually inspected 640 alarms from the 180 groups above in two different settings: inspection of the grouped alarms using the proposed method, and inspection of ungrouped common-POI alarms. In this evaluation, since the same set of alarms are to be inspected in two different settings, bias towards the second setting gets introduced due to the learning effect from the first setting. To avoid bias getting introduced to the proposed method, inspection of alarms using the proposed method was performed first: reduction in inspection effort observed through this ordering of the settings will be smaller than the reduction observed through reversed order of the settings. In both settings, the alarms were inspected partition-wise, and the measured time is *clock time*. This experiment indicated that the proposed method saves around 60% of the time required to manually inspect *common-POI alarms*. The reduction in the manual inspection time is achieved at the expense of the increase in the computation time needed to perform analysis and create sub-groups (column *Time Taken* in Table 5.3).

5.5.2 Evaluation of the Reuse-based AFPE Efficiency Improvement

Implementation To evaluate the reuse-based technique proposed to improve efficiency of AFPE, we implemented a series of state-of-the-art techniques that have been proposed for AFPE. We first performed grouping of related assertions [34] to reduce the overall number of model checking calls by processing related assertions together. For scalability of model checking, the code is pruned for assertions in each group using backward slicing [211] (called *partition-level slicing*). The code slices generated for each group are then processed using techniques that over-approximate loops whose bound cannot be determined statically [35, 46]. The assertions in the over-approximated code are verified using the context expansion approach [153, 174]. We used CBMC [28] as the model checker to verify the assertions. Before making a model checking call in the context of a function, the code is pruned (sliced) considering that function as the entry-point [45] (called *function-level slicing*). When a model checking call is skipped, the corresponding *function-level slicing* also can be skipped. Henceforth, we use *model checking call* to mean the both: call first to a slicer for function-level slicing and then to a model checker.

Selection of applications and alarms Table 5.5 presents five partitioned-code applications that we selected. The selection of applications was based on whether (a) alarms generated on them were common-POI alarms, and (b) we could run the implemented AFPE-techniques on them. The first two applications are selected from the benchmark used to evaluate our technique proposed to postprocess delta alarms in Chapter 6 (Section 6.8.1.1). The other three applications are selected from the benchmarks used to evaluate our proposed alarms repositioning techniques in Chapter 3 (Section 3.6.1.2) and Chapter 4 (Section 4.6.1).

We analyzed the partitions of the selected applications using TCS ECA [197] for three verification properties: *array index out of bounds* (AIOB), *division by zero* (DZ), and *overflow underflow (OFUF)*. Table 5.5 presents the number of alarms generated on those applications (column *Total Alarms*).

Table 5.5: Experimental results for evaluation of the reuse-based technique to improve efficiency of AFPE techniques applied to alarms generated on partitioned code.

Application	Partitions	Property	Total Alarms	Groups of related assertions	False positives eliminated	Total time for AFPE (in minutes)			Total model checking calls		
						Without reuse	With Reuse	% reduction in time	Without Reuse	With Reuse	% reduction in calls
smp_utils	29	AIOB	178	43	20	866.4	807.3	6.8	88	9	10.2
		OFUF	1534	277	198	1000.2	995.2	0.5	210	186	11.4
dict_gcide	8	AIOB	106	26	1	22.0	9.71	55.9	47	21	55.3
		DZ	128	100	8	23.2	13.75	40.6	94	39	58.5
		OFUF	652	331	43	40.3	32.4	19.7	139	92	33.8
uucp (5)	5	AIOB	22	9	0	66.4	45.3	31.7	6	4	33.3
		OFUF	343	136	0	213.65	170.9	20.0	23	18	21.7
ffmpeg (93)	93	AIOB	1775	507	340	904.2	805.0	11.0	589	549	6.8
		DZ	582	134	5	211.9	176.3	16.8	192	154	19.8
		OFUF	12375	1979	1314	4217.4	3909.7	7.3	1849	1721	6.9
Industry app (3)	3	AIOB	145	38	52	236.8	232.6	1.8	120	109	9.2
		DZ	9	6	0	69.1	69.1	0	18	18	0
		OFUF	300	89	75	452.8	442.1	2.4	208	175	15.9

Experimental Results Table 5.5 also presents the results of postprocessing the selected alarms using the implemented series of techniques. It shows the number of groups of related assertions formed, and the number of false positives eliminated using the implemented techniques. For each model checking call the time out threshold was set to 10 minutes. In the earlier studies [34, 45, 153] the time out threshold varies from two minutes to eight minutes. To increase the likelihood of alarms being verified or refuted, we preferred to increase the threshold compared to the earlier studies.

To evaluate our proposed reuse-based technique, we processed the alarms in two settings: with and without reusing AFPE results of common-POI alarms across partitions. The results indicate that, the proposed reuse of AFPE results across partitions reduces the number of model checking calls by up to 58.5%, with median reduction of 19.8%. The reduction in model checking calls reduced the total AFPE time by up to 56%, with median reduction of 12.15%. Note that, higher reduction in the number of model checking calls does not imply a similar reduction in the time: in several instances, the reduction in time is much smaller than the reduction in the number of calls. This occurs due to the following two reasons.

1. In general, the model checking calls which are made for higher contexts result in time-out, and those contexts are not common to multiple clusters and therefore their results are not reused. A call that results in time-out takes much more time (10 minutes), whereas the other calls generally take less than a minute.

2. The *total AFPE time* includes the time required to process all the alarms using AFPE techniques: common-POI as well as unique alarms. There is no reuse of results for unique alarms. Moreover, the *total AFPE time* also includes the time to generate *partition-level slices* and to over-approximate the loops. Our proposed reuse-based technique does not reduce the time taken by these two techniques.

Due to the above two reasons, the reduction in AFPE time seems to be lower than the reduction in effort to manually inspect common-POI alarms, where the effort reduction is measured for common-POI alarms only.

Like any other empirical study, the evaluations of the proposed alarms inspection method (Section 5.5.1) and reuse-based technique (Section 5.5.2) are subject to threats to validity. Since these evaluations are on similar lines to evaluations in Chapters 3, 4 and 6, we discuss the threats to validity of all these studies in the conclusions chapter (Section 7.3).

5.6 Related Work

In this section, we first compare our proposed grouping-based method to inspect common-POI alarms, and then the reuse-based technique to reduce the time taken by AFPE, with the techniques that are related to them.

Grouping-based Inspection Method Among the six approaches proposed for postprocessing of alarms, *clustering* and *simplification of manual inspection* are relevant to our proposed grouping-based inspection of common-POI alarms. Sound clustering techniques group similar or related alarms together and identify dominant alarms for each group, so that only a few alarms from each group get inspected. Our method is on similar lines, because inspecting only one alarm in most (66%) of the groups is sufficient. However, these existing clustering techniques fail to group similar or common-POI alarms that are generated for different partitions. In our method, we group these common-POI alarms together. Therefore, the clustering techniques and our proposed grouping method are orthogonal, and can be used together to process alarms generated on partitioned code. The existing techniques proposed to simplify manual inspection of alarms employ different methods. However, none of them takes into account that the common-POI alarms are generated for the same POI. We find that, the existing techniques can be combined with our proposed inspection method to obtain better results.

Reuse-based AFPE Technique Existing techniques that are proposed to improve efficiency of model checking-based AFPE are relevant to our reuse-based technique proposed to improve AFPE efficiency. Chimdyalwar and Darke [34] used data and control dependencies to form groups of related assertions (alarms). As they verify assertions in each group together, instead of verifying each assertion separately, the overall number of model checking calls are reduced. In our prior work [152, 153], we proposed techniques to predict result of a given model checking call, and used the predicted results to skip a subset of model checking calls. Wang et al. [209] used program slicing to improve efficiency of model checking-based false positives elimination. Applying these AFPE techniques to common-POI alarms can help to improve efficiency, however they may still result in repetitive model checking calls as they do not take into account that they are generated for the same program points. Our proposed reuse-based technique reduces the number of model checking calls by reusing results of the repeated calls across multiple partitions. This indicates that existing techniques and our technique, being orthogonal to each other, can be used together.

5.7 Conclusion

In this chapter, we have addressed the problem of (1) repetitively inspecting the same code during manual inspection of common-POI alarms, and (2) making the same model checking calls multiple times during AFPE applied to common-POI alarms. We considered the special category of common-POI alarms for their postprocessing, because around half of the alarms generated on partitioned-code are common-POI alarms, and applying the existing postprocessing techniques to them incurs redundancy. We addressed the redundancy problem by taking into account that the common-POI alarms are generated for the same POI but for multiple partitions.

To reduce the redundancy in inspection of common-POI alarms, incurred due to repetitive inspection of the same code, we proposed their grouping. For each group, we reported tops of each of the strict-ORCCs computed for common-POI alarms in the group. Based on the tops of the group, we proposed a method to inspect the grouped alarms. Our evaluation of the method indicates that

> Grouping of common-POI alarms and inspecting them based on the tops computed for each group allows to eliminate alarms in 66% of the groups just by inspecting only one alarm from each group. Skipping inspection of the other alarms from these groups reduces 60% of the effort required to manually inspect common-POI alarms.

To reduce the redundancy in applying AFPE to common-POI alarms, incurred due to repetitive model checking calls made for common-POI alarms, we reused results of model checking calls across the partitions. The reuse of results allows to reduce the number of model checking calls, and thereby to improve AFPE efficiency. Our evaluation of the reuse-based technique indicates that,

> Reusing results of model checking calls across partitions for common-POI alarms allows to reduce the number of model checking calls by up to 58.5%, with median reduction of 19.8%. Skipping those repetitive model checking calls reduces *the total AFPE time* by up to 56%, with median reduction of 12.15%.

During evaluation of the reuse-based technique to improve AFPE efficiency, we observed that the context expansion approach does not take the hierarchy and structure of calling functions into account. Based on the hierarchy and structure of calling functions, we plan to reduce the number of model checking calls made during the context expansion approach.

The proposed assertions grouping technique [34] groups related alarms that belong to the same function. We observe that, during context expansion, model checking calls for related alarms belonging to different functions can be combined when the calls are made for common calling functions. Based on this observation, as future work, we plan to formulate and evaluate a technique to reduce the overall number of model checking calls.

Chapter 6

Postprocessing of Delta Alarms

Static analysis tools help to detect common programming errors but generate a large number of alarms indicating possible errors. Moreover, when applied to an evolving software system, many alarms are repeated from a previous version to the next one. Version-aware static analysis techniques *(VSATs) are proposed to reduce the number of alarms by suppressing the repeated alarms. The alarms reported by VSATs, i.e., the alarms remaining after the suppression, are called* delta alarms.

We observe that, postprocessing and reporting of delta alarms *can be further improved by taking into account code changes between the two versions. However, none of the existing VSATs postprocess delta alarms based on code changes. Based on this observation, we use code changes to (1) rank delta alarms, and (2) to improve efficiency of existing* automated false positive elimination *(AFPE) techniques when they are applied to delta alarms.*

We first classify delta alarms into six classes depending on the code changes generating them, and then rank the alarms by assigning different priorities to these classes. The ranking of alarms can help suppress alarms that are ranked lower when resources to inspect alarms are limited. Next, we postprocess ranked delta alarms for AFPE. We use the classes and code changes to determine situations where AFPE results from the previous version can be reused. The reuse of AFPE results across the two versions helps to improve efficiency of AFPE applied to delta alarms.

We performed an empirical evaluation using 9789 alarms generated on 59 versions of seven open source C applications. The evaluation results indicate that the proposed classification and ranking of delta alarms help to identify 61% of delta alarms as less likely to be errors than the others. The reuse of AFPE results across the versions reduces the number of model checking calls by median of 84.3%, which in turn reduces the AFPE time by median of 64.5%.

6.1 Introduction

In this section, we first introduce *version-aware static analysis techniques* (VSATs) and the notion of delta alarms generated by these VSATs. Next, we describe two problems associated with postprocessing of delta alarms. Then, we provide overview of solutions that we propose to address those problems.

6.1.1 Background

Static analysis tools help to automatically detect common programming errors like *division by zero* and *array index out of bounds* [12, 14, 21, 206] as well as to certify absence of such errors in safety-critical systems [24, 52, 112]. However, these tools are known to generate a large number of false alarms, i.e., spurious warning messages notifying the tool-user about potential errors [54, 92, 139, 181, 186]. Partitioning the alarms into true errors and false alarms (false positives) requires manual inspection [54, 120, 182]. The large number of false positives and effort required to manually analyze alarms have been identified as primary reasons for underuse of static analysis tools in practice [20, 37, 92, 120].

Furthermore, when applying a static analysis tool to evolving systems, numerous alarms reported by the tool for one version also get reported for the subsequent one. We call these alarms *repeated alarms*. A few of the alarms postprocessing techniques propose to reduce the number of repeated alarms by taking into account the code changes between the two versions (Section 2.4.3.2). We call these postprocessing techniques *version-aware static analysis techniques* (VSATs). Note that, VSATs are different from the *history-aware alarms ranking techniques* (Section 2.4.2.2) which analyze the software change history to rank alarms. The ranking is primarily based on identification of categories of alarms that are quickly or commonly fixed by the programmers. In contrast, VSATs aim to identify alarms that repeat across two subsequent versions and suppress them by taking into account the code changes between the two versions.

The alarms reported by VSATs, i.e., the alarms remaining after the suppression of repeated alarms, are called *delta alarms*. That is, a delta alarm reported by a VSAT is either newly generated alarm or a repeated alarm that is impacted by a code change between the two versions (called *impacted alarm*). Reporting of the latter type of alarm is required, because the alarm *may* be a new error due to a code change between the two versions.

Suppressing a repeated alarm based on the coding patterns [207] or syntactic location matching [191] can be unreliable: an alarm that indicates an error can get suppressed. Hence, more advanced reliable VSATs [36, 117, 133] have been proposed. Section 6.3 discusses the approaches used by these reliable VSATs for the safe suppression of alarms. Furthermore, it describes limitations of the reliable VSATs. In this chapter, we limit the discussion scope only to delta alarms reported by the reliable VSATs.

6.1.2 The Problem

Due to their limitations (discussed in Section 6.3), the reliable VSATs [36, 133] still report a large number of delta alarms: evaluations of reliable VSATs indicate that around 40-80% of the tool-generated alarms get reported as delta alarms. Therefore, delta alarms require further postprocessing to reduce their number and to simplify their manual inspection. We find that, in addition to *computing* delta alarms, the information about code changes can be used further to improve *postprocessing* of the alarms. However, *none of the VSATs further postprocesses delta alarms based on code changes.*

Moreover, we find that, since different program statements affect alarms differently [116], information about the changed program statements can be used to prioritize delta alarms so that lower ranked alarms can be suppressed when resources available to manually inspect them are limited. This observation led to the following research question.

> **RQ 6:** How can we rank delta alarms based on types of the code changes generating them such that the alarms ranked higher are more likely to be errors than the alarms ranked lower?

Next we consider applying *automated false positives elimination* (AFPE) to automatically reduce the number of delta alarms. As AFPE is known to perform poorly in terms of efficiency (discussed earlier in Section 5.4.1), improving its efficiency is important [34, 152, 153]. We find that, since impacted alarms are repeated across the two versions, the calls to a model checker made during AFPE applied to the impacted alarms are also repeated when the code previously verified by the model checker is not changed. Therefore, results of applying AFPE to alarms generated on the previous version can be reused when AFPE is applied to impacted alarms generated on the subsequent version. However, to the best of our knowledge, no alarms postprocessing technique reuses AFPE results across versions. This observation led to the following research question.

> **RQ 7:** How can we use code changes to improve efficiency of AFPE applied to delta alarms?

6.1.3 Overview of Our Solution

To address the problems discussed above, we propose postprocessing of delta alarms based on *the corresponding code changes*: the code changes due to which the delta alarms are generated. The proposed postprocessing has the following novelties.

Addressing RQ 6: We first classify delta alarms into six classes depending on type of the corresponding code changes. We then rank the delta alarms by assigning different priorities to these classes. The prioritization of the classes is based on our observation that the likelihood of a delta alarm being impacted by a code change generating it varies depending on the type of the change (and thus the class of the delta alarm). This observation is based on the finding by Kumar et al. [116] that many of the program statements appearing on the backward slice generated for an alarm do not affect whether the alarm is a false positive. Since the alarms in the lowest priority class(es) are most likely to be false impacted alarms, they can be suppressed. The alarms suppression may result in unsoundness (suppressing a genuine error), however it is unavoidable when the resources (time) available to manually inspect all the alarms are not sufficient.

Addressing RQ 7: Next, we use the code changes and the alarm classes to improve efficiency of AFPE applied to delta alarms. The efficiency improvement is obtained by reducing the number of model checking calls made during AFPE. We reduce the number of model checking calls by determining the instances where AFPE results from the previous version can be reused. The reuse of results is for impacted alarms and determined based on whether the code being verified by the model checking calls is changed between the two versions.

We design our techniques, proposed for the above postprocessing, to accept the following inputs: (1) delta alarms generated on the current version being analyzed using static analysis, (2) results of static analysis and AFPE on the previous version, and (3) mapping of the source code of the previous version to the current version (Section 6.4). This design allows postprocessing delta alarms independently of the VSATs generating them and techniques used to identify changes between the two versions.

We performed an empirical evaluation of the proposed technique using 9789 alarms generated by a commercial static analysis tool, TCS ECA [197], on 59 versions of seven open source C applications. The evaluation results indicate that, the proposed classification and ranking of delta alarms help to identify 61% of delta alarms as less likely to be errors than the others.

The reuse of AFPE results across the versions reduces the number of model checking calls by median of 84.3%, which in turn reduces the analysis time by 64.5%.

The following are key contributions of this chapter.

1. A technique to classify delta alarms into six classes based on the code changes generating them, and rank the delta alarms by prioritizing those classes.
2. A reuse-based technique to improve efficiency of AFPE applied to delta alarms.
3. An empirical evaluation of the techniques above using 9789 alarms generated on 59 versions of seven open source C applications.

Chapter Outline Section 6.2 presents the terms and notations that we use throughout this chapter. Section 6.3 describes limitations of the reliable VSATs. Section 6.4 describes the prerequisites (inputs) for our technique. Section 6.5 describes our classification of delta alarms, while Section 6.6 presents the proposed ranking. Section 6.7 describes the reuse-based technique proposed to improve efficiency of AFPE. Section 6.8 discusses our empirical evaluation. Section 6.9 presents related work, and Section 6.10 concludes.

6.2 Background: Terms and Notations

We use the terms and notations described in Sections 3.2.1 and 4.2. Following we describe a few additional terms and notations that we use in this chapter.

6.2.1 Data and Control Dependencies

Recall that data dependencies of a variable are the definitions on which the variable is data dependent (Section 4.2.1). For a reaching definition d_v of v (i.e., a data dependency of v), we use *assignExpr*(d_v) to denote the assignment expression at d_v: the expression having an assignment to v at d_v. We say that data dependencies of an assignment expression e are same as union of data dependencies of variables in RHS of e. For an expression other than assignment, its data dependencies are defined as the union of data dependencies of variables in it.

Recall that control dependencies of a node n are the conditional edges on which n is control dependent (Section 4.2.1). For a given control dependency $u \rightarrow v$, we use *label*$(u \rightarrow v)$ to denote its label, and use *condExpr*$(u \rightarrow v)$ to denote the *conditional expression* associated with the branching node u. When a conditional edge $u \rightarrow v$ 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*. We

say that control dependencies of an expression e are same as control dependencies of the node corresponding to e.

For a control dependency (conditional edge) e, data dependencies of e are same as data dependencies of *condExpr*(e). Control dependencies of a data dependency $d_x : x = expr$ (resp. a control dependency $u \rightarrow v$) are same as the control dependencies of node corresponding to d_x (resp. node u).

Let α be a variable/expression at a program point p, or a data/control dependency. We write $d \xrightarrow{\text{d}} \alpha$ (resp. $e \xrightarrow{\text{c}} \alpha$) to denote d is a data dependency of α (resp. e is a control dependency of α). A reaching definition (assignment expression) d is a *transitive data dependency* of α if $d_1 \xrightarrow{\text{d}} d_2 \xrightarrow{\text{d}} d_3 \xrightarrow{\text{d}} ... \xrightarrow{\text{d}} d_k$, where $d_1 = d,\ d_k = \alpha,\ d_i \xrightarrow{\text{d}} d_{i+1}$, and $k \geq 2$. We use $d \xrightarrow{\text{d+}} \alpha$ to denote that d is a transitive data dependency of α. Let D be the set of all possible data dependencies (definitions of variables) in the program. We denote transitive closure of data dependencies of α using $dDep^{+}(\alpha)$, where $dDep^{+}(\alpha) = \{d \mid d \in D,\ d \xrightarrow{\text{d+}} \alpha\}$.

On similar lines, a conditional edge e is a *transitive control dependency* of α (shown as $e \xrightarrow{\text{c+}} \alpha$) if $e_1 \xrightarrow{\text{c}} e_2 \xrightarrow{\text{c}} e_3 \xrightarrow{\text{c}} ... \xrightarrow{\text{c}} e_k$, where $e_1 = e,\ e_k = \alpha,\ e_i \xrightarrow{\text{c}} e_{i+1}$, and $k \geq 2$. We write $cdDep \xrightarrow{\text{cd}} \alpha$ to denote $cdDep$ is a data or control dependency of α. A dependency, i.e., a reaching definition or a conditional edge, $cdDep$ is a *transitive data and control dependency* of α (shown as $cdDep \xrightarrow{\text{cd+}} \alpha$) if $cd_1 \xrightarrow{\text{cd}} cd_2 \xrightarrow{\text{cd}} cd_3 \xrightarrow{\text{cd}} ... \xrightarrow{\text{cd}} cd_k$, where $cd_1 = cdDep,\ cd_k = \alpha,\ cd_i \xrightarrow{\text{cd}} cd_{i+1}$, and $k \geq 2$.

Let C be the set of all possible control dependencies (conditional edges) in the program. We denote transitive closure of data and control dependencies of α using $cdDep^{+}(\alpha)$, where $cdDep^{+}(\alpha) = \{d \mid d \in D,\ d \xrightarrow{\text{cd+}} \alpha\} \cup \{e \mid e \in C,\ e \xrightarrow{\text{cd+}} \alpha\}$.

6.2.2 Program Slicing

For given a program and a set of variable(s) at a program point of interest, *program slicing* [211] computes a program that contains only those statements that are likely to influence the values of the variables at that program point [116]. The computed program is called a *program slice*. Depending on the use of program slice, several *backward* slicing techniques have been proposed [189, 199], such as backward slice [211], and thin slice [192]. Backward slice [211] consists of program statements that correspond to both data and control dependencies, whereas thin slice [192] consists of the statements that correspond only to data dependencies.

Kumar et al. [116] have recently proposed the notion of *value slice*, which is a pruned version of the 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 the values of variables in e. We call the control dependencies on a value slice *value dependencies*. In other words, a value slice is obtained by eliminating from the backward slice the control dependencies and their transitive dependencies, that only decide whether the program point of e is reachable. For example, consider the expression $arr[x]$ at line 8 in V_1 in Figure 6.1. Its control dependency $n_7 \rightarrow n_8$ is not a *value dependency*, however the same control dependency is a value dependency for the expression y present at line 11.

We use $e \xrightarrow{\text{v}} \alpha$ to denote that a conditional edge e is a value dependency of α. A conditional edge e is a *transitive value dependency* of α (shown as $e \xrightarrow{\text{v+}} \alpha$) if $e_1 \xrightarrow{\text{v}} e_2 \xrightarrow{\text{v}} e_3 \xrightarrow{\text{v}} ... \xrightarrow{\text{v}} e_k$, where $e_1 = e,\ e_k = \alpha,\ e_i \xrightarrow{\text{v}} e_{i+1}$, and $k \geq 2$. We use $vdDep \xrightarrow{\text{vd}} \alpha$ to denote $vdDep$ is a data or value dependency of α. A dependency $vdDep$ is a *transitive data and value dependency* of α

Version V_1

```
1   int foo(){
2     int x = 0, y = 0, arr[10];
3
4     x = lib();
5
6
7     if(nondet()){
8       y = arr[x];        A'8
9     }
10
11    x = y;
12  }
```

Version V_2

```
1   int foo(){
2     int x = 0, y = 0, arr[12];
3
4     x = lib();
5
6     x = x + 2;
7     if(nondet()){
8       y = arr[x];        A8
9     }
10
11    x = y;
12  }
```

Figure 6.1: Illustrating limitation of IA-VSAT: it reports A_8 as a delta alarm while VMV suppresses the alarm.

if $vd_1 \xrightarrow{\text{vd}} vd_2 \xrightarrow{\text{vd}} vd_3 \xrightarrow{\text{vd}} ... \xrightarrow{\text{vd}} vd_k$, where $vd_1 = vdDep,\ vd_k = \alpha,\ vd_i \xrightarrow{\text{vd}} vd_{i+1}$, and $k \geq 2$. We write $vdDep \xrightarrow{\text{vd+}} \alpha$ to denote that $vdDep$ is a transitive data and value dependency of α. We denote transitive closure of data and value dependencies of α using $vdDep^{+}(\alpha)$, where $vdDep^{+}(\alpha) = \{d \mid d \in D,\ d \xrightarrow{\text{vd+}} \alpha\} \cup \{e \mid e \in C,\ e \xrightarrow{\text{vd+}} \alpha\}$.

The three slices—backward, value, and thin—generated for an expression are such that backward slice subsumes[1] value slice, and value slice subsumes thin slice: 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 [116, 192]. The techniques to generate these three slices create a *program dependence graph* [64] to determine which dependencies are to be retained on the slice and which ones to be removed. We say that a program slice created for e is same as the slice created together for the variables in e.

6.2.3 Static Analysis Alarms

Recall that an expression that is checked by a static analysis tool is called a *point of interest* (POI) (Section 1). For example, a POI for check related to DZ and AIOB respectively corresponds to a denominator and access of an array with its index. We use $poi(\phi)$ to denote the POI of an alarm ϕ, and $\phi^{p}_{l,V}$ to denote an alarm reported for a POI of verification property p, present in line l of version V. We say that a slice generated for an alarm ϕ is same as the slice generated for $poi(\phi)$.

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, $0 \leq x \leq 11$ is the alarm condition of A_8 shown in Figure 6.1. Note that the alarm condition of an alarm is a logical formula, independent of the program point of the alarm. We say that alarms ϕ and ϕ' of the same property are *similar* if $cond(\phi) \Leftrightarrow cond(\phi')$.

We assume that a static analysis tool groups the generated alarms using state-of-the-art clustering techniques [123, 156], 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.

[1] Slice X subsumes slice Y iff every dependency in Y is also present in X.

6.3 Background: VSATs and Their Limitations

In this section, we briefly describe the existing *version-aware static analysis techniques* (VSATs), classify them into two classes, and present their limitations. Henceforth in this chapter, we use V_1 and V_2 to denote two subsequent code versions.

6.3.1 VSATs and Their Classification

The approaches used by the existing VSATs [36, 117, 133, 191, 207] to suppress repeated alarms vary greatly, and include such methods as *syntactic location matching* [191], *coding patterns* [191], and *impact analysis* [36]. Based on whether the techniques can result in suppression of an error (false negative), we classify them into the following two classes.

1. **Unreliable VSATs:** The techniques [191, 207] of this class can result in suppressing an error, because they are based on *syntactic location matching* [191] and *coding patterns* [207]. Therefore, we call these techniques *unreliable VSATs*: soundness guarantees cannot be provided for the false alarms suppressed by them. As these techniques are unreliable, we exclude them from subsequent discussion.

2. **Reliable VSATs:** The techniques [36, 117, 133] in this class perform *safe suppression* of repeated alarms, i.e., they do not suppress a true error. Therefore, we call these techniques *reliable VSATs*: they provide soundness guarantees for the suppressed false alarms. The safe suppression of alarms is based on an 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 errors. The approaches used by these reliable techniques [36, 117, 133] include *impact analysis* [36], *carrying over semantic information from the previous version* [133], and *differential assertion checking* [117]. We exclude the differential assertion checking-based VSAT [117] from our discussion in this chapter, because it checks two versions of a program with respect to a set of assertions (alarms), i.e., it does not consider newly generated alarms. Below we describe the approaches used by the other two VSATs.

   (a) The VSAT proposed by Chimdyalwar and Kumar [36], performs an *impact analysis* to identify code in V_2 impacted by the code change between V_1 and V_2, and suppresses alarms that are generated on V_2 and not impacted by the changes. We call this VSAT *impact analysis-based VSAT* (IA-VSAT). IA-VSAT first creates program dependence graphs for the repeated alarms on both the versions V_1 and V_2, and uses the graphs to compute alarms that are impacted by the code changes between V_1 and V_2.

   (b) The VSAT proposed by Logozzo et al. [133], also called as *verification modulo versions* (VMV), first extracts semantic environment conditions—sufficient or necessary conditions—from V_1, and instruments the code in V_2 to insert those conditions at the corresponding locations. Later, the instrumented code is verified, where the conditions suppress a subset of alarms generated on V_2 by proving them as safe.

6.3.2 Limitations of Reliable VSATs

The above two VSATs have fundamentally different strengths and limitations. To illustrate this, consider the C code shown in Figure 6.1, where the code changes between the versions V_1 and V_2 are highlighted. Analyzing each of the versions using a static analysis tool for AIOB generates

Version V_1

```
1     int foo(){
2       int a, b, t;
3       if (a <= 100)
4         a = 100 - a;
5       else
6         a = a - 100;
7
8       while (a >= 1){
9         a = a - 1;
10        b = b - 2;
11      }
12
13      return 1 / b;        D'13
14  }
```

Version V_2

```
1     int foo(){
2       int a, b, t = 0;
3       if (a <= 100)
4         a = 100 - a;
5       else
6         a = a - 100;
7       print(t);;
8       while (a >= 1){
9         a = a - 1;
10        b = b - 2;
11      }
12
13      return 1 / b;        D13
14  }
```

Figure 6.2: Illustrating limitation of VMV: it reports D_{13} as a delta alarm while IA-VSAT suppresses the alarm.

an alarm, respectively shown as A'_8 and A_8. Ideally, a VSAT should suppress A_8 because if A'_8 is a false positive, A_8 is also a false positive. However, IA-VSAT [36] when applied on V_2 reports A_8 as a delta alarm, because the changes at lines 3 and 7 are found to be impacting the alarm: the data dependency of $poi(A_8)$ is changed. In contrast, VMV [133] suppresses the same alarm, because it first extracts $0 \leq x \leq 9$ as a *sufficient condition* at line 4, and accordingly instruments the code in V_2. Later, while analyzing the instrumented code, the same array access ($arr[x]$) at line 8 is proved safe, i.e., no alarm is generated for the array access.

As another example, consider the two versions shown in Figure 6.2 (adapted from [96]). Analyzing these versions using a static analysis tool for DZ property generates an alarm, respectively shown as D'_{13} and D_{13}. Ideally, a VSAT should suppress D_{13} as it is not impacted by the changes. IA-VSAT suppresses D_{13}. However, inferring a useful correctness condition—sufficient or necessary condition—corresponding to D_{13} as required by VMV is challenging [96]. When a useful correctness condition cannot be computed, VMV reports the alarm as a delta alarm, i.e., in Figure 6.2 D_{13} is reported as a delta alarm.

Due to their different strengths and limitations and they being reliable, the above two VSATs 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. Consider the example shown in Figure 6.3. Analyzing the code in V_1 and V_2 using a static analysis tool for AIOB and DZ properties generates *four* and *six* alarms respectively. These alarms are shown using rectangles. None of these two reliable VSATs or their combination suppresses any of the six alarms generated on V_2. Therefore, all the alarms generated on V_2 get reported as delta alarms.

We use the example in Figure 6.3 as the running example for the rest of this chapter.

In addition to the limitation discussed above, the reliable VSATs report false delta alarms when the code from the two subsequent versions cannot be mapped precisely due to code refactoring and movement (semantics preserving changes). Thus, the VSATs still report a large number of delta alarms: evaluations of the reliable VSATs indicate that around 40-80% of the generated alarms get reported as delta alarms. Hence, postprocessing of delta alarms is required [36, 133]. Furthermore, to benefit from multiple VSATs and the different techniques available to map the

Version V_1

```
1   int x, y, z, a[50];
2   int arr[10] = {...};
3
4   void foo(int p){
5     int m = 2, t;
6
7     y = lib2();
8     z = lib3();
9
10    if(nondet())
11      m = 4;
12    bar(m);
13
14
15    t = x / y;                    D'15
16  }
17
18  void bar(int i){
19    int t = 0, t1;
20
21    x = lib1();
22    t1 = arr[x];                  A'22
23
24    if(z > 20)
25      t = arr[y];                 A'25
26
27    a[i] = 5;
28
29    print(25 / t);                D'29
30  }
```

Version V_2

```
1   int x, y, z, a[50];
2   int arr[10] = {...};
3
4   void foo(int p){
5     int m = 2, t;
6
7     y = lib2();
8     z = lib4();
9
10    if(nondet())
11      m = z;
12    bar(m);
13    z = 1000 / p;          D13   //POI-added
14
15    t = x / (y - 2);       D15   //POI-changed
16  }
17
18  void bar(int i){
19    int t = 0, t1;
20
21    x = lib3();
22    t1 = arr[x];           A22   //Impacted
23
24    if(z > 20)
25      t = arr[y];          A25   //Impacted
26
27    a[i] = 5;              A27   //Result-changed
28
29    print(25 / t);         D29   //Impacted
30  }
```

Figure 6.3: Examples of delta alarms (generated on V_2).

code in two versions [61, 134], the postprocessing of delta alarms ought to be independent of the VSATs and code mapping techniques.

6.4 Pre-requisites (Inputs) for Our Technique

To postprocess delta alarms independently of VSATs and code mapping techniques, we designed our technique to accept the following inputs: delta alarms, mapping of the code in V_1 to V_2 (called code mapping), and static analysis results on V_1. The input delta alarms can be generated by any VSAT, reliable or unreliable. Below we describe the code mapping taken as input.

We assume that a mapping function Map_{V_1,V_2}: $\mathit{lines}(V_1) \rightarrow \mathit{lines}(V_2) \cup \{\perp\}$ is given, which maps source code lines in V_1 to their corresponding lines in V_2, and to $\perp$ if the lines have been deleted from V_1. Moreover, no two lines in V_1 map to the same line in V_2. We use this map to compute the following.

(i) A line l_1 in V_1 is *deleted* if $\text{Map}_{V_1,V_2}(l_1) = \perp$.

(ii) A line l_2 is *added* in V_2 if there does not exist l_1 in V_1 such that $\text{Map}_{V_1,V_2}(l_1) = l_2$.

(iii) A line l_1 in V_1 (or l_2 in V_2) is *changed* if $\text{Map}_{V_1,V_2}(l_1) = l_2$ and the code on l_1 and l_2, excluding the white spaces, is different.

(iv) A line l_2 in V_2 (or l_2 in V_2) is *unchanged* if $\text{Map}_{V_1,V_2}(l_1) = l_2$ and the code on l_1 and l_2, excluding the white spaces, is same.

When $\text{Map}_{V_1,V_2}(l_1) = l_2$ and $l_2 \neq \perp$, we say that l_1 and l_2 are *corresponding lines*. For a *changed* line l_1 in V_1 and its corresponding (changed) line l_2 in V_2, similar to mapping of the 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 $\perp$ if the token has been deleted from l_1. Similar 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_{l_1,l_2}: $\textit{tokens}(l_1) \rightarrow \textit{tokens}(l_2) \cup \{\perp\}$ to denote the mapping of tokens in l_1 to their corresponding tokens in l_2. Similar 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_{V_1,V_2}, and the mapping of tokens in changed lines, we compute the following.

(i) An expression e_1 at line l_1 in V_1 is *deleted* if (1) l_1 is deleted from V_1, or (2) l_1 is changed and every token in e_1 is deleted from l_1.

(ii) An expression e_2 is added to line l_2 in V_2 if (1) l_2 is added to V_2, or (2) l_2 is changed and every token in e_2 is added to l_2.

(iii) 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.

(iv) 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.

(v) 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 [97]. Moreover, we assume that on each line, there exists at most one program statement or a part of it.

6.5 Classification of Delta Alarms

Existing VSATs [36, 133, 191, 207] broadly classify delta alarms only into two classes: newly generated and impacted. For completeness, we define these classes while we describe our classification of delta alarms into six classes. The classification is based on code changes generating the alarms, where the alarms-generating code changes are identified based on (1) the code mapping taken as input (Section 6.4), (2) analysis results on the previous version, and (3) program dependence graphs generated for the alarms on both the versions.

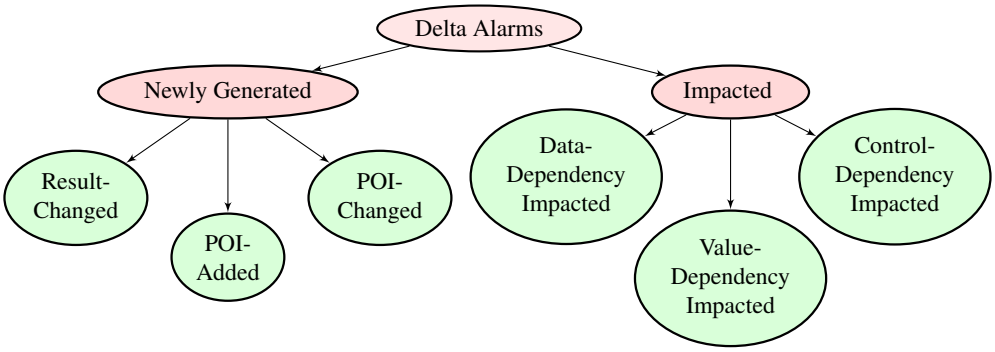


Figure 6.4: Classification of delta alarms into six classes.

6.5.1 Intuition Behind Our Classification of Delta Alarms

As changes between the versions V_1 and V_2 are of different types, so are the delta alarms generated due to these changes. Intuitively, these alarms should be processed differently as per the type of the changes generating them. We believe that classifying delta alarms based on the type of code changes is a more natural way to classify delta alarms, because it captures the causal relationship, and such a classification can provide multiple benefits. For example, a delta alarm that is newly generated because its POI got added in V_2 can be differentiated from another newly generated alarm whose POI already existed in V_1 but the same POI was safe in V_1. Based on the reasons for their generation, these two types of alarms can be respectively classified into two classes, e.g., *poi-added* and *analysis result changed*. Later, the process to inspect alarms in these two classes can vary: for the alarms in the former class, generally the complete code on its backward slice needs to be inspected, whereas for the alarms in the latter class, inspecting only the changed code that impacts the alarm is sufficient. Moreover, such a classification of delta alarms can help us identify classes that are more important than the others. For example, impacted delta alarms generated due to code changes that are *more likely to be semantically equivalent* can be deemed to be *less important* than the other impacted alarms.

Based on the observations above, we classify delta alarms into six classes shown in Figure 6.4. Following we describe these classes. We use the alarms shown Figure 6.3 to provide examples of alarms in the classes.

6.5.2 Classification of Newly Generated Delta Alarms

The newly generated alarms are the ones which did not occur in the previous version.

Definition 6.5.1 (Newly Generated Delta Alarm)**.** An alarm $\phi^p_{l_2,v_2}$ is called a *newly generated delta alarm* if l_2 is added in V_2, or line l_1 in V_1 and l_2 are the corresponding lines and no similar alarm was reported for l_1.

For example, D_{13}, D_{15}, and A_{27} are *newly generated* delta alarms on V_2.

The existing VSATs do not further classify the newly generated alarms. In our technique, we classify these alarms into three sub-classes, namely *result-changed, POI-changed*, and *POI-added*. These sub-classes are defined below.

6.5.2.1 Result-Changed Delta Alarm

Definition 6.5.2 (Result-Changed Delta Alarm)**.** We call a newly generated delta alarm $\phi^p_{l_2,V_2}$ a *result-changed delta 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 delta alarm, its POI also exists in the earlier version and analysis result for the POI is changed from *safe* in V_1 to *an alarm* in V_2. For example, A_{27} is a result-changed delta alarm, because the corresponding POI in V_1 is safe. This change in the result is due to the change at line 11.

6.5.2.2 POI-Changed Delta Alarms

Definition 6.5.3 (POI-Changed Delta Alarm)**.** We call a newly generated alarm $\phi^p_{l_2,V_2}$ a *POI-changed delta 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, D_{15} is a POI-changed delta alarm, because its POI $y - 2$ is changed from y in V_1 (i.e., y at line 15 in V_1 is the corresponding expression of the alarm's POI), and an alarm of the same type was reported for the corresponding expression y in V_1.

6.5.2.3 POI-Added Delta Alarms

Definition 6.5.4 (POI-Added Delta Alarm)**.** We call a newly generated alarm $\phi^p_{l_2,V_2}$ a *POI-added delta 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, D_{13} is a POI-added delta alarm, because its line 13 is added in V_2.

6.5.3 Classification of Impacted Alarms

As discussed in Section 6.3.2, the existing VSATs use different techniques to compute repeated alarms which can be suppressed. In general, the repeated alarms that cannot be suppressed by a VSAT are called *impacted alarms*.

For a given expression $expr$ in version V, let $cdDep^+(expr, V)$ be the set of transitive closure of data and control dependencies of $expr$. That is, transitive data and control dependencies in $cdDep^+(expr, V)$ correspond to program statements which appear in the backward slice generated for $expr$, and vice versa. Henceforth, we use dependency of $expr$ to refer to a data or control dependency that appears in the transitive closure of data and control dependencies of $expr$.

Definition 6.5.5 (Modified Dependencies)**.** We call a dependency d of an expression $expr$ in V_1 (resp. V_2) a *modified dependency* if the following holds:

1. d is a data dependency (definition) and *assignExpr*(d) is deleted (resp. added) or changed; or
2. d is a control dependency (conditional edge) and *label*$(u \to v)$ is changed, or *condExpr*$(u \to v)$ is deleted (resp. added) or changed. ■

Definition 6.5.6 (Impacted Delta Alarm)**.** An alarm $\phi^p_{l_2,V_2}$ is called an *impacted delta alarm* if

(i) *poi*($\phi^p_{l_2,V_2}$) is unchanged, and a similar alarm $\phi^p_{l_1,V_1}$ was reported for the corresponding (same) POI in V_1; *and*

(ii) at least one of the dependencies in $cdDep^+(poi(\phi^p_{l_2,V_2}), V_2)$ or $cdDep^+(poi(\phi^p_{l_1,V_1}), V_1)$ is modified. ■

Note that presence of a modified dependency is also checked in $cdDep^+(poi(\phi^p_{l_1,V_1}), V_1)$ (Case ii), because checking the presence of a modified dependency only in $cdDep^+(poi(\phi^p_{l_2,V_2}), V_2)$ does not capture deletion of a data dependency from V_1. For each impacted delta alarm $\phi^p_{l_2,V_2}$, there exists unique similar alarm $\phi^p_{l_1,V_1}$ corresponding to it where l_1 and l_2 are the corresponding lines. We call these two alarms, $\phi^p_{l_2,V_2}$ and $\phi^p_{l_1,V_1}$, *corresponding alarms*.

None of the existing VSATs further classify the impacted alarms. In our technique, we classify those alarms into three sub-classes, namely *data-dependency impacted alarms, value-dependency impacted alarms*, and *control-dependency impacted alarms*. These sub-classes are defined below.

6.5.3.1 Data-dependency Impacted Alarms

For a given expression $expr$ in version V, let $dDep^+(expr, V)$ be the transitive closure of data dependencies of $expr$.

Definition 6.5.7 (Data-dependency Impacted Alarm)**.** Let $\phi^p_{l_2,V_2}$ be an impacted alarm, with $\phi^p_{l_1,V_1}$ as its corresponding alarm. We call $\phi^p_{l_2,V_2}$ a *data-dependency impacted alarm* if at least one of the data dependencies in $dDep^+(poi(\phi^p_{l_2,V_2}), V_2)$ or $dDep^+(poi(\phi^p_{l_1,V_1}), V_1)$ is modified. ■

The dependencies in $dDep^+(expr, V)$ correspond to the program statements which appear in the thin slice generated for $expr$ in V, and vice versa. Therefore, in other words, an impacted alarm $\phi^p_{l_2,V_2}$ is a data-dependency impacted alarm if the thin slice generated for it is different from the thin slice generated for its corresponding alarm on V_1. For example, A_{22} is a data-dependency impacted alarm, because the data dependency of x at line 21 is modified.

6.5.3.2 Value-Dependency Impacted Alarms

For a given expression $expr$ in version V, let $vdDep^+(expr, V)$ be the transitive closure of data and value dependencies of $expr$.

Definition 6.5.8 (Value-Dependency Impacted Alarm)**.** Let $\phi^p_{l_2,V_2}$ be an impacted alarm, with $\phi^p_{l_1,V_1}$ as its corresponding alarm. We call $\phi^p_{l_2,V_2}$ as a *value-dependency impacted alarm* if

1. there exists a modified dependency in $vdDep^+(poi(\phi^p_{l_2,V_2}), V_2)$, but the same modified dependency is not present in $dDep^+(poi(\phi^p_{l_2,V_2}), V_2)$; or

2. there exists a modified dependency in $vdDep^+(poi(\phi^p_{l_1,V_1}), V_1)$, but the same modified dependency is not present in $dDep^+(poi(\phi^p_{l_1,V_1}), V_1)$. ■

The dependencies in $vdDep^+(expr, V)$ correspond to the program statements which appear in the value slice generated for $expr$ in V, and vice versa. Therefore, in other words, an impacted alarm $\phi^p_{l_2,V_2}$ is a value-dependency impacted alarm if the thin slices generated for it and its corresponding alarm on V_1 are same, but their value slices are different. For example, D_{29}

is a value-dependency impacted alarm, because (1) its value slice is different from the value slice generated for its corresponding alarm D'_{29}, but the thin slices of the two alarms are the same. Their value slices are different, because the data dependency $z = lib4()$ at line 8, which is present in $vdDep^{+}(poi(D_{29}))$, is modified. Their thin slices are the same because this modified dependency does not appear in $dDep^{+}(poi(D_{29}))$ and no other data dependency in them is modified.

6.5.3.3 Control-Dependency Impacted Alarms

Definition 6.5.9 (Control-Dependency Impacted Alarm)**.** Let $\phi^{p}_{l_2,V_2}$ be an impacted alarm, with $\phi^{p}_{l_1,V_1}$ as its corresponding alarm. We call $\phi^{p}_{l_2,V_2}$ as a *control-dependency impacted alarm* if

1. there exists a modified dependency in $cdDep^{+}(poi(\phi^{p}_{l_2,V_2}), V_2)$, but the same modified dependency is not present in $vdDep^{+}(poi(\phi^{p}_{l_2,V_2}), V_2)$; or

2. there exists a modified dependency in $cdDep^{+}(poi(\phi^{p}_{l_1,V_1}), V_1)$, but the same modified dependency is not present in $vdDep^{+}(poi(\phi^{p}_{l_1,V_1}), V_1)$. ■

In other words, an impacted alarm $\phi^{p}_{l_2,V_2}$ is a control-dependency impacted alarm if the value slices generated for the alarm and its corresponding alarm on V_1 are same, but their backward slices are different. For example, A_{25} is a control-dependency impacted alarm, because the data dependency $z = lib4()$ at line 8, which is present in $cdDep^{+}(poi(A_{25}))$, is modified. Moreover, this modified dependency is not present in $vdDep^{+}(poi(A_{25}))$, and the value slices of the two corresponding alarms are same.

We use the above six classes to (1) rank or suppress alarms based on anecdotal evidence (next section); and (2) improve efficiency of AFPE (Sect. 6.7).

6.6 Ranking of Delta Alarms

In this section, we describe ranking of delta alarms, that we obtain by prioritizing the six classes (discussed in the previous section). Like the existing alarms ranking techniques, our ranking of delta alarms helps to (1) suppress low priority alarms when the available resources to manually inspect all the alarms are limited; and (2) select first the alarms that are more likely to be errors (even when all of them can be inspected).

6.6.1 Prioritization of Newly Generated and Impacted Alarms

We make the following observations for the two main classes of the delta alarms: newly generated and impacted.

6.6.1.1 Newly Generated Alarms

VSATs suppress each alarm that repeats in V_2 when the alarm is not impacted by the changes between V_1 and V_2. Thus, if a newly generated delta alarm is suppressed, the alarm will remain suppressed on the subsequent versions, unless a code change between the next two subsequent versions impacts the alarm. That is, the newly generated alarms are reported for the POIs that are either added or changed as a part of the changes between the two versions. Thus, these alarms

are directly related to the changes as opposed to the impacted alarms that are indirectly generated due to changes in their transitive data and control dependencies.

6.6.1.2 Impacted Alarms

The changes made between V_1 and V_2 generally correspond to fixing of bugs, addition of features, and refactoring. Failure to detect refactorings can result in generation of *false* impacted alarms. Moreover, determining whether a change (made to fix a bug or add a feature) impacts an expression is undecidable in general [178]. Hence the VSATs use conservative impact analysis that is based on data and control dependencies. As a result, very often an expression gets (falsely) identified as impacted [76], and in turn a large number of *false* impacted alarms are generated.

6.6.1.3 Prioritization of Newly generated and Impacted Alarms

Based on the observations above, we prioritize newly generated alarms over impacted alarms. Next, we describe ranking of alarms in each of these two main classes, obtained by prioritizing their sub-classes.

6.6.2 Ranking of Newly Generated Alarms

We rank newly generated alarms by assigning different priorities to their sub-classes. The prioritized sub-classes are *Result-changed* $>$ *POI-added* $>$ *POI-changed*. This prioritization is based on the following observations.

1. For a result-changed alarm, its corresponding POI in V_1 was reported as safe however the same POI is an alarm in V_2. The change in the analysis result is more likely to be due to the side-effect of code changes and thus we believe that such alarms are to be inspected on a higher priority.

2. We prioritize POI-added alarms over POI-changed alarms, because POIs of the former alarms are newly added in the code whereas POIs of latter alarms are changed from POIs existing in V_1, and the newly generated alarms whose POIs are added are to be given higher priority than the alarms whose POIs existed in the previous version (discussed above in Section 6.6.1).

6.6.3 Prioritization of Classes of Impacted Alarms

Our prioritization of impacted alarms is based on the evaluation performed by Kumar et al. [116] for the three comparable slices—thin, value, and backward slices. In their evaluation, Kumar et al. observed that using the thin slice instead of the backward slice affects 29% of the alarms. That is, removing the transitive control dependencies from backward slice affected 29% of the alarms: the alarms got changed from false positives to errors. For the other 71% alarms, the removal of the dependencies did not affect the alarms. Therefore, for a majority of alarms, their transitive control dependencies do not affect the alarms (and in such cases only the transitive data dependencies affect the alarms). Moreover, in the cases where the transitive control dependencies affect alarms, the transitive data dependencies also affect the alarms. As a consequence of this, we make the following observation.

> A change made to an impacted alarm's transitive data dependency is more likely to impact the alarm, compared to a change that directly or indirectly impacts the alarm's transitive control dependency. Since data-dependency impacted alarms are generated due to a change in at least one of their transitive data dependencies, compared to the other alarms, they are more likely to get impacted by the corresponding code changes.

Moreover, in their experiments, Kumar et al. [116] observed that using the value slice instead of the backward slice affects only 2% of the alarms. That is, removing the control dependencies from backward slice, that are not value dependencies, affects only 2% of the alarms, and adding the value dependencies of the alarms to their thin slices reduced the affected alarms from 29% to 2%. Therefore, for most of the alarms, their control dependencies that are not value dependencies are actually *not relevant* for them, whereas the value dependencies often are actually relevant for the alarms. As a consequence of this, we can say that, when a change directly or indirectly impacts an impacted alarm's value dependency, the change is more likely to impact the alarm, compared to the change that directly or indirectly impacts the alarm's control dependency that is not a value dependency. Value-dependency impacted alarms are generated due to changes of the former type, whereas control-dependency impacted alarms are generated due to changes of the latter type. Thus, compared to value-dependency impacted alarms, control-dependency impacted alarms are less likely to get impacted by the corresponding code changes, i.e., they are more likely to be false positives.

Based on the discussion above, we propose the following prioritization for the sub-classes of impacted alarms:

Data-dependency impacted > *Value-dependency impacted* > *Control-dependency impacted*.

6.6.4 Grouping of Same-class Alarms

As alarms in a sub-class of impacted alarms still can be a large in number, we group together impacted alarms that are generated due to the same modified dependencies. With this grouping, the grouped alarms—that are generated due to the same reason(s)—get inspected together.

6.7 Improving Efficiency of AFPE

In this section, we first recapitulate the problem of poor efficiency of model checking-based AFPE. Then we describe a class of model checking calls that are made for impacted alarms and repeat across the two versions. We refer to these calls *repeated model checking calls*. Last, we propose a technique to improve AFPE efficiency by identifying and skipping those repeated model checking calls.

6.7.1 Recapitulation: The Problem of Poor AFPE Efficiency

Recall the discussion in Section 5.4.1. Existing AFPE techniques [34, 152, 153, 174] generate an assertion corresponding to each alarm and use model checking to verify the assertions. An alarm is eliminated as a false positive when its corresponding assertion holds. The techniques employ context expansion approach [174] to use a model checker in a more scalable way. However, the approach considerably increases the number of model checking calls, and hence, further degrades efficiency of AFPE.

Grouping of related assertions [34, 152] has been proposed to verify multiple related assertions together, and thus to reduce the number of model checking calls. However, the number of generated groups of related assertions is still large, and the context expansion approach gets applied to each group. Evaluations of the context expansion-based AFPE techniques [34, 45, 152, 153] indicate that processing an assertion or a group of related assertions, on average, results in making five model checking calls and takes around three to four minutes. Due to the large number of model checking calls and each call taking considerable amount of time, applying AFPE to alarms becomes time-consuming and ultimately renders AFPE unsuitable for postprocessing of alarms generated on large systems. For example, processing 200 groups of related alarms would require more than 10 hours.

6.7.2 Terms and Notations

Recall from Section 5.4.1 that the verification context in a model checking call is provided as a function. Therefore, we use the terms *context* and *function* interchangeably. We use *mcall*(Φ, f) to denote a model checking call that verifies a group of assertions generated for alarms Φ and in the context of function f.

We use *slice*(Φ, f) to denote backward slice generated for a set of alarms Φ and in the context of f, i.e., with f as the entry function. Note that while the *declarations of global variables* appearing on the sliced code are outside the scope of f, they are still included in the sliced code, otherwise the sliced code would not compile. We say that two slices are *identical* iff every statement in one slice has a corresponding identical statement in the other slice, with the ordering of the statements preserved; and vice versa.

6.7.3 Repeated Model Checking Calls for Impacted Alarms

We find that, model checking calls made during AFPE applied to impacted alarms can be same as the calls made during AFPE applied to their corresponding alarms on the previous version. We refer to such calls as *repeated model checking calls*. These calls repeat across the two versions because the code change(s) that generate an impacted alarm belong to only a few contexts (functions), while majority of the contexts in which the corresponding assertion is to be verified are unchanged. We illustrate this observation by means of Figure 6.3. Assume that the alarms generated on the earlier version V_1 have been processed using AFPE techniques. Consider the *three alarms generated on* V_1 for the POIs in *bar*. Performing grouping of related assertions generated for those alarms, based on their data dependencies, results in two groups: $\{A'_{22}\}$ and $\{A'_{25}, D'_{29}\}$. Verifying each group using context expansion results in making two model checking calls per group: first in the context of *bar* and then in the context of *foo*.

Next, consider the four delta alarms generated on V_2 for the POIs in *bar*. Performing grouping of related assertions generated for those alarms, based on their data dependencies, results in three groups: $\{A_{22}\}$, $\{A_{25}, D_{29}\}$, and $\{A_{27}\}$. Verifying each group using context expansion results in making two model checking calls per group: first in the context of *bar* and then in the context of *foo*.

Consider the second group of assertions, $\{A_{25}, D_{29}\}$, generated for impacted alarms A_{25} and D_{29}. The first model checking call made for this group is *mcall*$(\{A_{25}, D_{29}\}, bar)$. Note that, between the two versions, there is only one change in *bar* (at line 21), and this change does not impact the two alarms. As a result, *slice*$(\{A_{25}, D_{29}\}, bar)$ and *slice*$(\{A'_{25}, D'_{29}\}, bar)$ are identical, and results of the two calls, *mcall*$(\{A_{25}, D_{29}\}, bar)$ and *mcall*$(\{A'_{25}, D'_{29}\}, bar)$, ought to be the same. Therefore, *mcall*$(\{A_{25}, D_{29}\}, bar)$ is a *repeated model checking call*

compared to the model checking calls made during AFPE applied on V_1. During AFPE applied to alarms generated on V_2, if such repeated model checking calls are identified, the results of applying AFPE to their corresponding alarms on V_1 can be *safely* reused *instead of making the repeated calls again*. Skipping those repeated calls improves the efficiency of AFPE applied to alarms on V_2.

Note that, the second model checking call for the same group, *mcall*$(\{A_{25}, D_{29}\}, foo)$ cannot be identified as a repeated call, because the two slices, *slice*$(\{A_{25}, D_{29}\}, foo)$ and *slice*$(\{A'_{25}, D'_{29}\}, foo)$ are not identical. On similar lines, the first and second model checking calls made for $\{A_{22}\}$, i.e., *mcall*$(\{A_{22}\}, bar)$ and *mcall*$(\{A_{22}\}, foo)$, are not repeated calls because the corresponding slices in the two versions are not identical. Moreover, none of the model checking calls for $\{A_{27}\}$ is repeated, because A_{27} is a newly generated alarm which does not have a corresponding alarm on V_1 and hence no corresponding model checking calls.

Summarizing, among the six model checking calls required for delta alarms generated on V_2, *mcall*$(\{A_{25}, D_{29}\}, bar)$ can be identified as repeated and skipped.

6.7.4 Our Solution

As discussed earlier, AFPE using context expansion approach results in a large number of model checking calls (Sections 6.7.1), and some of them can be *repeated* (Section 6.7.3). Recall that the repeated calls exist for impacted alarms only: for newly generated alarms, there are no model checking calls for their corresponding alarms on the previous version. In our evaluation of the technique for ranking and classification of delta alarms, we found that around 87% of delta alarms are *impacted*, while the remaining 13% are newly generated (Section 6.8.1.2). Considering the large number of impacted alarms, we design a technique for identifying and skipping redundant model checking calls. As discussed above in Section 6.7.3, for a set of delta alarms Φ, identification of a model checking call *mcall*(Φ, f) as redundant requires taking into account (1) whether the alarms in Φ are impacted or newly generated, and (2) whether f is changed during changes between the two versions.

For an impacted alarm ϕ, we use *corrAlarm*(ϕ) to denote its corresponding alarm on the previous version. On similar lines, for a set of impacted alarms Φ, we use *corrAlarms*(Φ) to denote the set of their corresponding alarms on the previous version.

Let Φ_g be a set of impacted alarms such that the assertions generated for them form a group of related assertions. Recall that a model checking call *mcall*(Φ_g, f) is repeated iff *slice*(Φ_g, f) is identical with *slice*(*corrAlarms*$(\Phi_g), f)$. Note that, the assertions generated for *corrAlarms*(Φ_g) on the previous version need not be in the same group. For example, the call *mcall*$(\{A_{25}, D_{29}\}, bar)$ would still be a repeated call, even if assertions generated for the alarms on V_1 are not in the same group. This is because, the computation of repeated model checking calls depends on whether the verification context is changed, and not on how the assertions were verified (together or individually).

6.7.4.1 Eliminating the Need for Comparison of the Slices

The existing techniques prune the code using program slicing before each model checking call: *mcall*(Φ, f) verifies *slice*(Φ, f). When a model checking call is repeated, the slicing before it is redundant. Hence, skipping slicing for such repeated model checking calls also can help to improve AFPE efficiency. Based on this observation, we design our technique to compute repeated model checking calls without computing slices.

Recall that a delta alarm ϕ is identified as an impacted alarm if at least one of the dependencies in $cdDep^{+}(poi(\phi), V_2)$ or $cdDep^{+}(poi(\phi), V_1)$ is *modified* (Section 6.5.3). During classification of delta alarms, for each identified impacted alarm, we store the *modified dependencies* in those two PDGs separately. We call the modified dependencies computed for an impacted alarm ϕ *modified dependencies of* ϕ. We use $modDeps(\phi)$ to denote the modified dependencies in $cdDep^{+}(poi(\phi), V_2)$, and use $modDeps(corrAlarm(\phi))$ to denote the modified dependencies in $cdDep^{+}(poi(corrAlarm(\phi)), V_1)$. Therefore, the modified dependencies of an impacted alarm ϕ are given as $modDeps(\phi) \cup modDeps(corrAlarm(\phi))$.

Let f^* denote the set of functions, that includes f and the functions that are directly or indirectly called by f. We use $funcs(modDeps(\phi))$ to denote the functions in which the modified dependencies of an impacted alarm ϕ appear. These functions appear on the backward slice of ϕ and are changed between the two versions. We use these functions to compute whether $slice(\{\phi\}, f)$ and $slice(corrAlarms(\{\phi\}), f)$ are identical. These slices are identical only if

1. on V_2, $funcs(modDeps(\phi)) \cap f^* = \emptyset$, and
2. on V_1, $funcs(modDeps(corrAlarm(\phi))) \cap f^* = \emptyset$.

In other words, if in both the versions, none of the functions in f^* has a statement which corresponds to a modified dependency of the alarm, the two slices are identical. Using this approach, for a set of impacted alarms Φ, we identify a model checking call $mcall(\Phi, f)$ as repeated iff for every alarm $\phi \in \Phi$, the above two conditions hold.

Note that, determining whether the two slices are identical as described above does not take into account changes made to declarations of global variables. The declarations of global variables are not a part of any function. A change in the declaration of a variable (its data type) on those slices can result in different behaviors at the program points of the corresponding alarms. Consequently, results of verifying those two slices can be different, and hence a call identified as repeated using the approach above is not necessarily a repeated call. Therefore, the approach above computes *over-approximation* of the repeated calls. The over-approximation might result in a failure to eliminate false positives which could have been eliminated otherwise. We expect such cases to be rare.

We illustrate misidentification of repeated calls by referring to the alarms shown in Figure 6.5. Due to the changes at lines 3 and 7, A_{19} is an impacted alarm. The code in the functions *bar* and *foo* is not changed: statement corresponding to a modified dependency of this alarm does not belong to these two functions. Therefore, for this impacted alarm, our approach to compute repeated model checking calls identifies the two model checking calls, $mcall(\{A_{19}\}, bar)$ and $mcall(\{A_{19}\}, foo)$, as repeated. The result of $mcall(\{A'_{19}\}, foo)$ is *counter-example*, and it will be reused for $mcall(\{A_{19}\}, foo)$. However, due to the change at line 3, the result of $mcall(\{A_{19}\}, foo)$ is different from that of $mcall(\{A'_{19}\}, foo)$: the index expression never gets evaluated to a value higher than 399. Since the result of the skipped (repeated) call $mcall(\{A_{19}\}, foo)$ is *counter-example*, the context expansion approach requires making the next call $mcall(\{A_{19}\}, main)$. There are two possibilities for $mcall(\{A_{19}\}, main)$: either it verifies the assertion, or results in *time out* (TO)/*out of memory* (OM). In the former case, the assertion will be eliminated as a false positive although the call $mcall(\{A_{19}\}, foo)$ is misidentified as a repeated call. In the latter case, or in the absence of the caller function *main*, the alarm will not be identified and eliminated as a false positive.

In summary, although the above approach to compute repeated model checking calls allows to eliminate the need to explicitly compute and compare slices in the corresponding verification contexts, the approach can result in misidentification of repeated calls. Expecting that such cases

Version V_1

```
1   const int arr[]={0,13,28,46};
2   int largeArr[400];
3   unsigned int var;
4
5   int main(){
6     ...
7     foo(lib1());
8   }
9
10  void foo(int p){
11    unsigned int i= lib2();
12    unsigned int j= lib3();
13    if (i < 5 && j < i && p > 0)
14      bar(arr[i]-arr[j]);
15  }
16
17  int bar(int n){
18    var = lib3();
19    largeArr[n + var] = 0;     A'19
20  }
```

Version V_2

```
1   const int arr[]={0,13,28,46};
2   int largeArr[400];
3   unsigned char var;
4
5   int main(){
6     ...
7     foo(lib4());
8   }
9
10  void foo(int p){
11    unsigned int i= lib2();
12    unsigned int j= lib3();
13    if (i < 5 && j < i && p > 0)
14      bar(arr[i]-arr[j]);
15  }
16
17  int bar(int n){
18    var = lib3();
19    largeArr[n + var] = 0;     A19
20  }
```

Figure 6.5: Example of an impacted alarm to illustrate over-approximation of computation of repeated model checking calls.

would be rare in practice, we employ the approach above to compute over-approximated repeated model checking calls. When failure to eliminate a false positive during AFPE is not to be allowed, the repeated model checking can be computed by explicit computation and comparison of those slices, which can reduce the gain in AFPE efficiency. This indicates trade-off between *precision* (the number of false positives eliminated) and *efficiency* of AFPE.

6.7.4.2 Proposed Approach

Let Φ be a set of impacted alarms. The approach described above to compute whether a given model checking $mcall(\Phi, f)$ is repeated, is based on checking whether *the context of f is changed*, i.e., whether $slice(\Phi, f)$ and $slice(corrAlarms(\Phi), f)$ are identical. Note that when the context of f is changed, we conservatively assumed that, due to the change, the verification result of $mcall(\Phi, f)$ can be different from the verification results of $mcall(corrAlarms(\Phi), f)$. We call this approach *conservative approach*.

The time taken by a model checking call that times out depends on the *time threshold* specified by the user, which is usually between two to eight minutes [34, 45, 152, 153], whereas the other types of calls take, on average, around 30 seconds. Hence, on large applications, a high percentage of AFPE time is taken by the model checking calls which result in *time out*. Identifying in advance the calls which result in time-outs and skipping them will provide more efficiency gain than the gain obtained by skipping an equal number of other calls. Based on this observation, we aim to reduce the number of model checking calls that result in time out. To this end, we propose an *aggressive* variant of the approach described above. This approach is based on expectation that change(s) made to an existing function are minor, and do not significantly affect its complexity. Thus, for an impacted alarm ϕ, when result of $mcall(corrAlarm(\phi), f)$ is TO, the call $mcall(\{\phi\}, f)$ is most likely to result in TO. Consequently, for a set of impacted

Table 6.1: Summary of the two approaches proposed to compute whether a given model checking call $mcall(\Phi, f)$ is repeated.

Status of delta alarms, (Φ)	Context change status for f (computed based on comparing the slices)	AFPE results of the corresponding alarms on the earlier version	Computation of repeated model checking calls	
			Conservative approach	Aggressive approach
All are impacted	Not changed	-	**Repeated**	**Repeated**
	Changed	Contains at least one TO or OM	Not repeated	**Repeated**
		All are counter-examples	Not repeated	**Repeated**
		All other cases	Not repeated	Not repeated
At least one is new, and at least one impacted	Not changed	Contains at least one TO or OM	Not repeated	**Repeated**
		Does not contain TO or OM	Not repeated	Not repeated
	Changed	-	Not repeated	Not repeated
All are newly generated	-	-	Not repeated	Not repeated

TO = Time Out, OM = Out of Memory.

alarms Φ, if there exists $\phi \in \Phi$ such that $mcall(corrAlarm(\phi), f)$ results in TO, $mcall(\Phi, f)$ is also most likely to result in TO. Note that, this approach, unlike the conservative approach, also takes into account the verification results of assertions generated for the corresponding alarms on the previous version.

When the code being verified by a model checking call is complex, the call also can result in *out of memory* (OM) instead of TO. Therefore, we use the similar approach to determine and skip the model checking calls which are most likely to result in OM. In our aggressive approach, in addition to identifying these calls (TO and OM) as repeated, we identify the following calls as repeated: all the alarms in the group are impacted, verification context is changed, and the corresponding verification results on the previous version are counter-examples. In these three cases (TO, OM, and counter-examples), result of the verification call $mcall(\Phi, f)$ will be same for all the assertions generated corresponding to Φ: TO, OM, or counter-example.

In Table 6.1, we summarize and compare the two approaches. The table indicates the following:

1. Both the approaches identify a call $mcall(\Phi, f)$ as repeated when all the alarms in Φ are impacted, and the verification context f is not changed. (In this case, for the assertions generated corresponding to Φ, there is one to one mapping of the verification results across the versions).

2. Conservative (resp. aggressive) approach identifies a call $mcall(\Phi, f)$ as *not repeated* when all the alarms (resp. at least one alarm) in Φ are newly generated, irrespective of whether the context f is changed.

3. Aggressive approach identifies a call $mcall(\Phi, f)$ as repeated when the AFPE results on the previous version are counter-examples or contain at least one TO or OM, irrespective of whether the context f is changed.

Compared to the conservative approach, the aggressive approach identifies more model checking calls as repeated and skips them, but it may fail to eliminate false positives which could have been eliminated by the conservative approach.

6.8 Empirical Evaluation

In this section, we evaluate the proposed technique to classify and rank delta alarms (Section 6.6), and the technique to improve efficiency of AFPE applied to delta alarms (Section 6.7).

6.8.1 Evaluation of the Classification and Ranking Technique

We first describe the setup we used to evaluate the technique, and then discuss the evaluation results.

6.8.1.1 Experimental Setup

Implementation As a baseline, we used implementation of *impact analysis-based VSAT* proposed by Chimdyalwar and Kumar [36] (discussed in Section 6.3). The implementation is available in TCS ECA [197] that supports analysis of C programs. We implemented the delta alarms classification technique using the analysis framework of TCS ECA. The framework provides APIs to generate PDGs corresponding to backward, thin, and value slices, and accessing dependencies in the PDGs. In the implementation of *impact analysis-based VSAT*, *diff* is used to create a mapping of the code from two subsequent versions. We used the same code mapping as the input to our delta alarms classification technique.

Selection of Applications and Alarms Evaluation of the techniques presented in this chapter requires to analyze multiple versions of an application. We did not have access to multiple versions of the industry applications that we used in the earlier evaluations (Chapters 3, 4, and 5). Therefore, to evaluate these techniques, we used open source applications whose at least two versions are available online. Moreover, to limit the amount of analysis time, we restricted the evaluation to relatively small applications. We randomly chose seven open source C applications from the list of 100 applications used by Cha et al. [29], with the constraints that (1) application size shown in the list should be greater than 10 KLOC and lesser than 20 KLOC, and (2) at least two versions of the application should be available online. Table 6.2 lists these applications together with the total number of versions selected, and the first and last versions in our selection. In total, we analyzed 59 versions using TCS ECA for AIOB verification property. The analysis of the application versions selected, the computation of delta alarms, and our proposed postprocessing of the delta alarms is performed using a machine with i7 2.5GHz processor and 16GB RAM.

For each application, Table 6.2 summarizes the total number of tool-generated alarms (column *TCS ECA alarms*), and *delta alarms* generated on its selected versions except the first version, i.e., the number delta alarms generated on V_2 (compared to V_1) + the number of alarms generated on V_3 (compared to V_2) and so on. The alarms generated on the first version are not a part of this

Table 6.2: Experimental results showing the alarms in each of the classes of delta alarms.

Application	Details of the versions selected			TCS ECA alarms	Delta alarms	Newly generated alarms				Impacted alarms			
	Total version	First version	Last version			RC	PA	PC	Total	DD	VD	CD	Total
archimedes	15	0.0.8	2.0.0	6373	4183	0	328	15	343	215	578	3047	3840
auto-apt	3	0.3.22	0.3.23	178	174	0	0	0	0	0	84	90	174
dict-gcide	2	0.48.1	0.48.2	192	16	0	5	0	5	0	7	4	11
gzip	9	1.3.9	1.9	1514	1446	0	223	1	224	59	1118	45	1222
mtr	12	0.73	0.85	803	400	0	28	0	28	12	200	160	372
rhash	15	1.2.4	1.3.7	245	207	0	17	0	17	13	64	113	190
smp-utils	3	0.96	0.98	484	484	0	255	0	255	0	2	227	229
Grand Total				9789	6910	0	856	16	872	299	2053	3686	6038

RC = Result-changed, PA = POI-added, PC = POI-changed, DD = Data dependency impacted,
VD = Value dependency impacted, and CD = Control dependency impacted.

table, because analysis of this version is not version-aware: no previous version is available to suppress alarms generated on this version.

6.8.1.2 Experimental Results

Using the technique from Section 6.5, we classified delta alarms generated on the selected versions. Table 6.2 presents the number of alarms that belonged to each of the six classes proposed. Inspecting the table, we make the following observations.

- Around 70% of the tool-generated alarms get reported as delta alarms; the remaining alarms are suppressed by the VSAT.
- The impacted alarms dominate the newly generated alarms: around 87% delta alarms are impacted while the remaining 13% are newly generated.
- Majority (98%) of the newly generated alarms belong to POI-added class, whereas six out of the eight applications had no POI-changed alarms at all.
- Among the impacted alarms, only 5% are data dependency impacted alarms: only a small fraction of impacted alarms are generated due to changes in their transitive data dependencies.
- Among the impacted alarms, 34% and 61% respectively are value dependency impacted and control dependency impacted alarms.

Recall that, in the ranking scheme of Figure 6.4, result-changed alarms are assigned the highest priority. In the evaluation, no newly generated alarm is a result-changed alarm. A possible reason to this could be that these applications are well tested or actually no such error existed in these applications. Since the control dependency impacted alarms are most likely to

be false impacted alarms (Section 6.6.1.2), they can be suppressed. Thus, overall, the proposed ranking allows to identify around 61% of delta alarms (3686 out of 6038) as less likely to be errors than the others. Therefore, these alarms can be suppressed when the resources (time) available to manually inspect delta alarms are limited.

6.8.2 Evaluation of Improvement in AFPE Efficiency

We first describe the setup we used to evaluate the technique proposed for improving AFPE efficiency, and then discuss the evaluation results.

6.8.2.1 Experimental Setup

Implementation To evaluate the technique proposed to improve efficiency of AFPE, we implemented a series of state-of-the-art techniques that have been proposed for AFPE. This implementation is same as the one used to evaluate the technique for reuse of AFPE results across multiple partitions (Section 5.5.2). For the sake of completeness, we recapitulate main steps of the implementation. We first performed grouping of related assertions [34] to reduce the overall number of model checking calls by processing related assertions together. For scalability of model checking, the code is sliced with respect to assertions in each group using backward slicing [211] (*application-level slicing*). The code slices generated for each group are then processed using techniques that over-approximate loops whose bound cannot be determined statically [35, 46]. The assertions in the over-approximated code are verified using context expansion approach [153, 174]. We used CBMC [28] as the model checker to verify the assertions. Before making a model checking call in the context a function, the code is sliced considering that function as the entry-point [45] (*function-level slicing*). Therefore, when a model checking call is skipped due to it being identified as repeated, the corresponding *function-level slicing* also can be skipped. Henceforth, we use *model checking call* to mean both the call to a slicer for function-level slicing and the subsequent call to a model checker.

We implemented AFPE in three different settings:

1. *Original:* AFPE in the original setting, i.e., without applying any of the approaches to improve AFPE efficiency.
2. *Conservative:* AFPE in which repeated model checking calls are computed using the conservative approach.
3. *Aggressive:* AFPE in which repeated model checking calls are computed using the aggressive approach.

Selection of applications and alarms Table 6.3 presents the applications and their versions that we selected to evaluate the technique. We analyzed the versions of the selected applications using TCS ECA for AIOB verification property, computed delta alarms from alarms generated by TCS ECA, and performed grouping of related assertions generated for the *delta alarms*. The third column in the table presents the number of groups of related assertions on each version.

6.8.2.2 Experimental Results

We verified the related groups of assertions in the three settings implemented for AFPE: original, conservative, and aggressive. For the sake of consistency with the evaluation in Chapter 5, each

Table 6.3: Evaluation results of the technique to improve AFPE efficiency by identifying the repeated model checking calls across the two versions.

Application	Version	Groups of related assertions	Model checking calls			Total time taken (in seconds)			False positives eliminated		
			Original	% reduction		Original	% reduction		Original	Conservative	Aggressive
				Conservative	Aggressive		Conservative	Aggressive			
archimedes	0.1.0	109	128	40.6	87.5	3459	31.0	72.6	1	1	0
archimedes	0.1.1	111	139	86.3	94.2	3434	64.5	79.1	1	1	1
archimedes	0.1.2	112	134	84.3	95.5	3505	54.2	81.3	1	1	1
archimedes	0.1.3	109	131	92.4	100.0	3730	70.7	84.6	1	1	1
archimedes	0.1.4	109	131	92.4	100.0	3693	70.6	84.7	1	1	1
archimedes	0.7.0	112	134	91.8	99.3	3712	70.9	84.9	1	1	1
archimedes	1.0.0	112	134	100.0	100.0	3713	96.6	96.6	1	1	1
archimedes	1.2.0	105	127	62.2	75.6	3471	37.3	61.7	1	1	1
archimedes	1.5.0	110	132	99.2	100.0	3585	95.0	94.5	1	1	1
archimedes	2.0.0	137	165	74.5	80	4637	54.1	66.5	1	1	1
smp_utils	0.98	37	78	39.7	100.0	24807	3.5	98.6	27	26	26

model checking call was set to time out after 10 minutes. For AFPE performed in each setting, we computed (1) the number of model checking calls that were made, (2) the time taken, and (3) the number of false positives eliminated. The time taken includes the time required for the entire processing after the groups of related assertions are generated. Table 6.3 presents these results for each of the settings. The number of model checking calls made and time taken, in both conservative and aggressive settings, are shown as percentage of reduction compared to the original setting. The results indicate the following:

- The conservative approach reduces the number of model checking calls by up to 100%, with the median reduction of 86.3%. The reduction in model checking calls reduces the total AFPE time by up to 96.6%, with the median reduction of 64.5%.
- The aggressive approach reduces the number of model checking calls by up to 100%, with the median reduction of 99.3%. The reduction in model checking calls reduces the AFPE time by up to 98.6%, with the median reduction of 84.6%.

Only in one instance, aggressive setting fails to eliminate one false positive that has been eliminated by the original and conservative settings (archimedes 0.1.0 version). The number of false positives eliminated from alarms generated on *archimedes* application is much lower compared to *smp_utils* application. In our manual analysis, we found that a large number of model checking calls resulted in *out of memory* because the code has multiple large-size arrays and a large number of calculations involving variables of *floating point* data types.

The results on *smp_utils* indicate that both the conservative and aggressive settings eliminate one false positive less than the ones eliminated by the original setting. We found that, those two settings failed to eliminate the false positive due to over-approximation of computation of repeated model checking calls. In this particular case, the model checking call on the earlier version resulted in time out, and the call on the new version was identified as repeated, and

thus TO result was reused. However, the same call in the original setting results in verification success. The context of the verification is not changed but the size of a global array variable (Section 6.7.4.1). This indicates the impact of over-approximation of computation of repeated model checking calls.

Note that, higher reduction in the number of model checking calls does not imply similar reduction in the time: the reduction in time is smaller than the reduction in the number of calls. This occurs because the *total AFPE time* also includes the time to over-approximate loops. The proposed approach to repeated calls identification does not reduce the time required for approximation of the loops.

Like any other empirical study, the evaluations of the proposed delta alarms classification technique (Section 6.8.1) and technique to improve AFPE efficiency (Section 6.8.2) are subject to threats to validity. Since these evaluations are on similar lines to evaluations in Chapters 3, 4 and 5, we discuss the threats to validity of all these studies in Conclusions chapter (Section 7.3).

6.9 Related Work

In this chapter, we proposed two techniques: one to classify and rank delta alarms, and the other one to improve efficiency of AFPE. Thus, we compare them with the techniques which employ similar categories of the postprocessing approaches, namely ranking, pruning, and AFPE.

Ranking of Alarms The existing techniques to rank alarms employ different techniques such as statistical analysis [114], history of the bugs and alarms fixing, and even feedback from the user. Among them, the techniques that are based on history of fixing of alarms [103, 104] and bugs [212] 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. [86] have proposed a technique to rank alarms generated on evolving code. They compute a graph that concisely and precisely captures differences between the derivations of alarms produced by a static analysis tool before and after the change. Later, they perform Bayesian inference on the graph, which enables to rank alarms by likelihood of relevance to the change.

As the alarms in the sub-classes of impacted alarms are still large in number, they can be further ranked using the other ranking techniques.

Pruning of Alarms The techniques in this category classify alarms mainly into two classes, actionable and non-actionable ones [155]. The non-actionable alarms being more likely to be false positives, they are not reported to the users. The techniques vary based on the methods they employ to achieve the classification, and a majority of the techniques are based on machine learning [77, 221]. The version-aware static analysis techniques (VSATs) [36, 117, 133, 191, 207] also belong to this category as they suppress a subset of the alarms generated, calling them as non-impacting or not important. As discussed (Section 6.3), these VSATs use code changes between the two versions only to compute delta alarms but not to postprocess them further. 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.

Automated False Positives Elimination Techniques proposed for AFPE primarily address the issues associated with AFPE, namely poor-performance and non-scalability to large systems. The existing techniques proposed to improve efficiency of AFPE are based on grouping of the assertions [34, 152] or follow methods to predict result of a given model checking call [153]. Even though these techniques are applied, still a large number of model checking calls get made due to the context expansion. Our technique uses code change information to determine whether a call being made is repeated compared to calls made during AFPE on the previous version, so that result from the previous version can be reused and the call can be skipped. To the best of our knowledge, none of the existing techniques for AFPE efficiency improvement is applied in the context of evolving software or based on (any kind of) reuse of the results.

As discussed above, the proposed two techniques are orthogonal to the existing postprocessing techniques that implement the similar approaches. Thus, they can be combined with the existing techniques to obtain more benefits as compared to the benefits obtained by applying them individually.

6.10 Conclusion

In this chapter, based on our observation that the existing version-aware static analysis techniques do not use code changes to further postprocess delta alarms, we proposed two techniques to postprocess those delta alarms by taking into account the code changes. The first technique classifies delta alarms into six sub-classes based on the type of changes generating them. Then it ranks the alarms by assigning different priorities to the classes identified. The prioritization of the classes is based on empirical evaluation performed for value slice, which suggests that the three comparable slices—backward, value, and thin—have varying number of program statements and those statements affect the point of interest differently. Our evaluation of the proposed ranking technique, performed using 9789 alarms generated on 59 versions of seven open source C applications, indicates that

> The proposed classification and ranking of delta alarms help to identify 61% of delta alarms as less likely to be errors than the others.

In the postprocessing of delta alarms, we also targeted improving efficiency of AFPE applied to the alarms. The second technique identifies a subset of model checking calls that repeat across versions, and for the repeated calls, it reuses results of the corresponding calls in AFPE on the previous version. We have proposed computation of over-approximation of repeated calls to skip function-level slicing calls as well. To compute repeated calls, we presented two approaches: conservative and aggressive. The aggressive approach identifies more repeated calls, but may fail to eliminate some false positives which could have been eliminated otherwise. Our evaluation of the technique, using delta alarms generated on 11 versions of two applications, indicates that

The conservative approach to compute repeated model checking calls reduces the number of model checking calls by median of 84.3%, which in turn reduces the total AFPE time by 64.5%.

The aggressive approach reduces the number of model checking calls by median of 99.3%, which in turn reduces the total AFPE time by 84.6%.

The evaluation of the technique proposed for AFPE efficiency improvement is based on delta alarms generated for two applications. The evaluation results are encouraging. In the future, we plan to evaluate this technique on a large number of delta alarms generated on a variety of industry and open source applications.

Chapter 7

Conclusions

In this chapter, first we summarize the main contributions of this thesis. Then we discuss threats to validity of our findings, and possible directions for future research.

7.1 Contributions

Automated static analysis tools (ASATs) help to detect common programming errors like *division by zero* and *array index out of bounds* as well as to certify absence of such errors in safety-critical systems. However, these tools are known to generate a large number of false alarms (false positives). Partitioning those alarms into true errors and false positives requires manual inspection. The effort required to manually analyze the alarms and the large number of false positives among them have been identified as the primary reasons for underuse of ASATs in practice.

To address the problem of large number of false positives and the cost associated with their manual inspection, scientific literature has extensively studied the ways of improving precision of ASATs. However, given that verification problems are undecidable in general, reporting of false positives by these tools is inevitable. Furthermore, many times, the tools compromise on precision to achieve analysis scalability or improve performance, further worsening the number of false positives. *Postprocessing of alarms*—processing the alarms after they are generated—has been the alternative approach widely researched to address the problem of alarms and the inspection cost associated with them.

Throughout this thesis, we have discussed the topic of *postprocessing of alarms*. In Chapter 1, we asked the following main research question.

> **RQ:** How can we improve postprocessing of static analysis alarms?

To answer the main question above, we first had to understand state-of-the-art techniques and approaches that have been proposed for postprocessing of alarms. To this end, we asked the following research question.

RQ 1: What approaches have been proposed for postprocessing of alarms?

In Chapter 2, we answered the above research question by surveying techniques that have been proposed for postprocessing of alarms. We performed the survey by conducting a *systematic literature search*, in which we combined *keywords-based database search* and *snowballing*. The search resulted in identifying 130 relevant research papers. In our categorization of postprocessing approaches proposed in these papers, we identified six main categories of the approaches: clustering, ranking, pruning, automated false positives elimination (AFPE), combination of static and dynamic analyses, and simplification of manual inspection. We studied and summarized the merits and shortcomings of those categories to assist users and designers/developers of ASATs to make informed choices. We find that the identified approaches are complementary and provide an opportunity to combine them.

Since our work aims at verifying safety-critical systems in the industry setting, we studied the techniques that are applicable for code proving—*sound clustering*, *ranking*, *AFPE*, and *simplification of manual inspection*—and identified their limitations, listed below.

1. Clustering of alarms is commonly used to reduce the number of alarms. However, state-of-the-art clustering techniques fail to group similar alarms, e.g., when the similar alarms belong to different branches of an *if* statement (Section 3.2).

2. Partition-wise postprocessing of common-POI alarms—the alarms generated for the same POIs but belonging to multiple partitions—results in repetitive processing. For example, manual inspection of alarms and automated false positives elimination require performing the same activity, or a part of it, multiple times for a group of common-POI alarms. As a result of the repetitive processing, the postprocessing of alarms incurs redundancy. However, none of the existing postprocessing techniques is designed to eliminate the redundancy by taking into account that the alarms are generated for the same POI.

3. Although the *context expansion* approach has helped model checking-based AFPE to use a model checker in a more scalable way, the approach considerably increases the number of model checking calls. As a result, AFPE performs poorly in terms of efficiency. The poor-performance ultimately renders AFPE unsuitable for postprocessing of alarms generated for large systems.

4. State-of-the-art techniques proposed for version-aware static analysis of evolving software take code changes into account only to compute delta alarms but not to postprocess the alarms further. During further required postprocessing of delta alarms, taking into account the code changes due to which the alarms are generated, helps to improve the postprocessing. These improvements are missed by the existing postprocessing techniques.

The above limitations indicate areas of improvement for the existing postprocessing techniques. Our work in the thesis targets to overcome those limitations.

In our study of alarms clustering, we found that Limitation 1 discussed above is due to the traditional reporting of alarms at the program points where their POIs are located. This way of

reporting alarms restricts identifying fewer alarms as dominant for the alarms appearing in those limitation scenarios. Hence, we asked the following research question.

> **RQ 2:** How can we automatically group similar alarms that state-of-the-art alarms clustering techniques fail to group?

Since the limitations of alarms clustering techniques arise due to reporting alarms at the locations of POIs for which the alarms are generated, the solution to the problem required a novel approach to change the alarms reporting locations (*repositioning of alarms*). Therefore, motivated by the work of Gehrke et al. [71], in Chapter 3 we proposed a technique to reposition alarms. We designed the technique to *safely* reduce the number of alarms (primary goal) by merging two or more similar alarms at a new program point. We observed that, in addition to the reduction, repositioning of alarms also provides an opportunity to report the alarms closer to their cause points (secondary goal). This reporting helps to reduce code traversals that one has to perform during manual inspection of alarms.

Repositioning alarms for the two goals requires identifying repositioning locations such that the results are maximized. To systematically identify such locations, we presented a data flow analysis-based technique. We have evaluated the technique using 33,162 alarms generated by a commercial static analysis tool, TCS ECA, on 20 open source and industry applications. The evaluation results indicate that,

> *Repositioning of alarms helps to reduce the number of alarms up to 20%, with median reduction of 7.25%.*

Contrary to our expectations, the median reduction observed during the experimental evaluation is limited. To understand why the median reduction is limited, we studied the reasons due to which the upward repositioning is stopped. For each reason, we measured the number of instances in which the repositioning is stopped due to that reason. In this study we observed that, in the majority of the cases, the upward repositioning of alarms is stopped due to this conservative assumption made about controlling conditions of the alarms when the alarms appear in only one branch of the *if* statements. We found that the limited reduction in the number of alarms was primarily due to this conservative assumption. To overcome this limitation we asked the following research question.

> **RQ 3:** How can we improve the reduction in the number of alarms obtained by repositioning them?

During our study of the repositioning limitation cases, we found that a large number of controlling conditions which stop upward repositioning of alarms do not affect the alarms. Therefore, identifying each controlling condition of an alarm, either as *impacting* or *non-impacting*, can help to consider or ignore the effect of the condition during repositioning and, in turn, to reposition the alarm further upward in the code. The further repositioning can result merging this alarm with similar one(s), and reduce the overall number of alarms.

Based on this observation in Chapter 4 we designed a technique to classify each control dependency of an alarm either as *impacting* or *non-impacting*. We identified a transitive control dependency of an alarm as a *non-impacting control dependency* (NCD), only if it does not affect whether the alarm is an error. Since computation of NCDs of an alarm is undecidable in

general, we followed an *aggressive* approach to compute approximated NCDs of alarms. During repositioning alarms, considering the effect of their NCDs allows to merge more similar alarms together, and thus, to further reduce the number of alarms. Summarizing,

> *Computing* non-impacting control dependencies *of alarms and taking into account their effect during repositioning helps to improve the reduction obtained.*

We evaluated the *improved repositioning technique* 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. We also compared the improved repositioning technique with the *original repositioning* in which alarms are repositioned without taking into account the effect of NCDs of the alarms. The evaluation results indicate that,

> *Compared to the original repositioning technique, the improved repositioning technique reduces the number of alarms by up to 36.09% and with median reduction of 10.48%.*

Our approach to approximately compute NCDs of similar alarms is observation-based, and the computation is based on a group of similar alarms. This approximated approach is required because (1) computing whether a transitive control dependency of an alarm is an NCD requires first determining whether the alarm is an error; and (2) a high percentage of alarms resulting after applying the original repositioning technique are still similar, and therefore reducing their number is important. In the evaluation of the technique, we performed manual analysis of a subset of repositioned alarms to evaluate effectiveness of the approximated approach. In the analyzed cases, we found that the approximated approach helped to reduce the number of alarms by 70%. However, the approximation resulted in 2% of the repositioned alarms detecting spurious errors. Therefore,

> *Our approach to approximately compute NCDs of similar alarms is effective: the approximation helps to* safely *reduce the number of alarms while it detects only a few repositioned alarms as spurious errors.*

In Chapter 5, we turned our attention to the limitation that existing postprocessing techniques do not address the redundancy in postprocessing of common-POI alarms (Limitation 2). The redundancy primarily occurs due to performing the same processing repeatedly for common-POI alarms, e.g., manual inspection of alarms and AFPE. We focus on manual inspection of alarms and AFPE because the manual effort to inspect alarms is costly and poor performance of AFPE has been identified as Limitation 3.

To eliminate the redundancy in manual inspection and AFPE of common-POI alarms (discussed above), we asked the below research questions.

RQ 4: How can we reduce redundancy in manual inspection of common-POI alarms?

RQ 5: How can we reduce redundancy in AFPE applied to common-POI alarms?

To reduce the redundancy postprocessing of common-POI alarms, we postprocessed them by taking into account that they are generated for the same program point. The redundancy reduction required processing a group of common-POI alarms together. Therefore, we proposed

a method to group common-POI alarms and inspect them together. When a grouped alarm is found to be a false positive in the context of each of the functions identified for the group, all the grouped alarms are false positives. Since the proposed method eliminates the need to inspect each common-POI alarm individually, the method reduces the redundancy incurred during their manual inspection. Our evaluation of the method indicates that,

> *Grouping of common-POI alarms and inspecting them based on the functions identified for each group allows to eliminate alarms in 66% of the groups just by inspecting only one alarm from each group. Skipping inspection of the other alarms reduces 60% of the effort required to manually inspect common-POI alarms.*

On similar lines of manual inspection of common-POI alarms, applying model checking-based AFPE to common-POI alarms results in repetitive model checking calls. Therefore, to eliminate the redundancy incurred due to those repeated calls, we reused results of model checking calls across the partitions. The reuse of results for common-POI alarms reduces the number of model checking calls, and thereby reduces the time taken by AFPE. Our evaluation of the reuse-based technique indicates that,

> *Reusing results of model checking calls across partitions for common-POI alarms allows to reduce the number of model checking calls by up to 58.5%, with median reduction of 19.8%. Skipping those repetitive model checking calls reduces* the total AFPE time *by up to 56%, with median reduction of 12.15%.*

In Chapter 6, we proposed techniques to overcome the limitation of state-of-the-art techniques proposed for version-aware static analysis techniques (Limitation 4). These techniques use code changes between two versions only to compute delta alarms but not to process the alarms further. Based on prior evaluations of program slicing techniques, we found that different program statements affect alarms differently. Consequently, changes made in statements affect the alarms differently, and classes of those changes can be used to classify and prioritize the alarms. Based on this observation, we asked the following question.

RQ 6: How can we rank delta alarms based on types of the code changes generating them such that the alarms ranked higher are more likely to be errors than the alarms ranked lower?

Next, we revisited our aim to improve efficiency of AFPE applied to delta alarms (Limitation 3).

RQ 7: How can we use code changes to improve efficiency of AFPE applied to delta alarms?

In this chapter, we proposed two techniques to postprocess delta alarms by taking into account the code changes generating the alarms. First we classified delta alarms into six classes depending on the classes of the code changes generating them. Next, by assigning different priorities to the classes of changes, based on the type of changed statements, we ranked delta alarms. Our evaluation of the proposed ranking technique, performed using 9789 alarms generated on 59 versions of seven open source C applications, indicates that

The proposed classification and ranking of delta alarms help to identify 61% of delta alarms as less likely to be errors than the others.

Since impacted alarms are repeated across the two versions, the calls to a model checker made during AFPE applied to the impacted alarms are also repeated when the code previously verified by the model checker is not changed. Based on this observation, we proposed a technique to identify repeated model checking calls and reuse results of the corresponding calls from AFPE on the previous version. Skipping those repeated calls allows to reduce the time taken by AFPE. To further reduce time, we skipped slicing calls made before repeated model checking calls. This improvement required approximately computing repeated calls, i.e., computing the calls without actually comparing the code sliced in the verification contexts from the two versions. To compute repeated calls, we presented two approaches: *conservative* and *aggressive*. The aggressive approach identifies more repeated calls, but may fail to eliminate some false positives which could have been eliminated otherwise. We believe that, depending on the efficiency to be obtained, one of the two approaches can be selected. Our evaluation of the technique, using delta alarms generated on 11 versions of two applications, indicates that

The conservative approach to compute repeated model checking calls reduces the number of model checking calls by median of 84.3%, which in turn reduces the analysis time by 64.5%.

The aggressive approach reduces the number of model checking calls by median of 99.3%, which in turn reduces the analysis time by 84.6%.

7.2 Discussion

As summarized above, the contributions of this thesis include several techniques to improve postprocessing of alarms. We obtain improvements by considering three different perspectives: the reporting methodology of alarms (repositioning of alarms), the nature of applications being analyzed (partitioned-code), and the nature of analysis (version-aware static analysis). We have observed that complexity of the problem of alarms' postprocessing is not only related to theoretical limitations of static analysis, such as undecidability [118], but also to diversity of goals static analysis can be applied for, applications being analyzed, and programming languages.

- Postprocessing of alarms differs greatly depending on whether the purpose of the analysis is *bug finding* or *code proving* (Chapter 2). For example, techniques to postprocess alarms generated on safety-critical applications would be different from the techniques applicable for postprocessing of alarms generated on applications that are not safety-critical. Techniques applicable for postprocessing of alarms generated by *code proving tools* can be applied to alarms generated by *bug finding tools*, but the converse is not true. Depending on the analysis purpose, the techniques to be used for alarms postprocessing vary.

- For effective postprocessing of alarms, the techniques need to take into account the type of applications being analyzed. For example, techniques which are suitable to process alarms generated on evolving code (Chapter 6) are not suitable to process alarms generated on partitioned-code (Chapter 5), and vice versa. When an application belongs to multiple types, applying techniques which are applicable to only a particular type of applications are not sufficient, and the postprocessing incurs redundancy. For example, analyzing partitioned-code of an evolving application would require combining techniques

that take into account the nature of multiple versions (relative correctness) and multiple partitions.

- There exists a variety in applications due to the programming languages. Analysis of programs written in different languages have different characteristics. These characteristics make postprocessing of alarms much more challenging. For example, precise analysis of C programs suffers from imprecision due to the programming characteristics such as pointers and accessibility to hardware (memory) registers. Precise analysis of C++/Java programs requires to consider the programming features such as aliases, virtual functions, (multiple) inheritance, and templates. Postprocessing of alarms in general requires re-analysis of the program, and hence, all those characteristics of the language need to be considered in the implementation of postprocessing techniques. These characteristics can affect the results of the techniques. For example, in the evaluation of the improved repositioning technique (Chapter 4), that is based on non-impacting control dependencies, we find that the reduction in the number of alarms obtained on C and COBOL applications vary considerably. The reason for the difference in the results could be that the COBOL applications were banking applications where the business logic is implemented in a series of nested if conditions whereas the applications written in C were data and file processing applications. Moreover, C applications were rich in pointers whereas COBOL applications do not use pointers.

  In our evaluations of the techniques, we found that the results varied considerably based on the applications even if they are written in the same language. This indicates that some applications had more patterns of the alarms to which the techniques were applicable, as compared to the other applications.

Therefore, to improve postprocessing of alarms, research needs to be continued along multiple lines to support different factors like the analysis purposes, different programming paradigms, and types of applications. Wherever there is an opportunity, multiple techniques should be combined, e.g. postprocessing of alarms generated on partitioned-code of an evolving application.

7.3 Threats to Validity

This section discusses threats to validity of our findings from evaluations of the techniques presented in Chapters 3, 4, 5, and 6. Threats to validity of our findings in Chapter 2 are separately discussed in Section 2.3.3.

7.3.1 Construct Validity

Threats to construct validity concern how accurately we operationalize the notions of *efficiency of AFPE*, and and *manual inspection effort*.

We measured the AFPE efficiency gain obtained due to the proposed techniques (Chapters 5 and 6), by measuring difference in *clock-time* taken by the two settings: AFPE without the techniques, and AFPE with the techniques. The efficiency gain also can be measured in terms of the number of model checking calls reduced, and the number of alarms skipped from the processing. The measurement based on these different parameter(s) would result in different results. Since the user perceives a reduction in time taken as an improvement when comparing two techniques, we measured efficiency gain as the difference in time taken.

Measuring reduction in effort, due to the method proposed to inspect common-POI alarms, required to measure manual effort to inspect alarms (Section 5.3). We measured the inspection effort as the *clock-time taken* to manually inspect those alarms. The inspection effort also can be measured as the number of alarms inspected per unit of time, the number of lines of code (or functions) traversed, etc. During evaluation of existing techniques that have been proposed to reduce effort required for manual inspection of alarms, time taken has been used to measure the effort [54, 150]. On similar lines, we measured the inspection effort as the time taken.

7.3.2 Internal Validity

Threats to internal validity concern the extent to which the observations are correctly derived from the experimental data. In our experiments, threats to internal validity concern *selection of alarms* to evaluate the techniques, *implementation of the techniques*, and *measurement of manual inspection of effort*.

7.3.2.1 Selection of Alarms

The findings of the presented techniques depend on alarms selected in the evaluation. The alarms generated vary as per the verification properties, type of applications, and static analysis tools used. To mitigate bias towards a particular category of alarms, we selected a large number of alarms generated for five verification properties and on a variety of applications. While selecting the properties, we ensured that the properties are based on different underlying analyses. For example, the properties *array index out of bound* and *division by zero* require analyzing code to compute values of variables, while *illegal dereference of a pointer* requires performing *pointer analysis*. Moreover, the property *uninitialized variables* requires computing whether variables at a program point are defined along all paths, and this analysis is different from value and pointer analyses.

Alarms generated by ASATs vary considerably based on the type and precision of the tools. The alarms selected in our experiments are generated by a commercial static analysis tool, TCS ECA, as we had access to this tool and its analysis framework. Moreover, the author had prior experience in designing different analyses using the framework. Using alarms generated by a single static analysis tool as sample can pose a threat to the validity of our findings. Since our experiments are based on a large number of alarms, generated for a variety of properties, and different types of applications, we expect that the findings would not vary much if alarms generated by some other static analysis tool are processed.

Use of benchmarks has been recommended to strengthen research validity in the software engineering research [82, 214]. Wherever possible, we selected applications from earlier benchmarks used to evaluate similar postprocessing techniques. For example, during evaluation of the repositioning techniques (Chapters 3 and 4), we included open source applications that were used to evaluate state-of-the-art clustering techniques (Section 3.6.1.2). Moreover, we ensured that the selected applications included both industry and open source applications, except evaluation of the techniques that process delta alarms (Chapter 6). The reason for the latter is that we were unable to get access to multiple versions of industry applications. In this case, we randomly selected applications from the applications used by Cha et al. [29] in their evaluation, whose multiple versions were available online.

7.3.2.2 Implementation of the Techniques

The presented techniques to evaluate reduction in the number of alarms require re-analysis of the applications on which they are generated, except the one presented to manually inspect groups of common-POI alarms. Depending on how precisely the required analyses in the techniques are implemented, the reduction in number of alarms and the time taken by AFPE can vary. For example, in the implementation of the original repositioning technique (Section 3.6.1.1), we propagated the conditions anticipable at the function-entry to its caller only if the function is called from a single place. We put this restriction to reduce the number of iterations required for repositioning alarms. The reduction in iterations allows to reduce the overall analysis time. However, this restriction can miss opportunities to merge a few more similar alarms together and reduce the number of alarms. Therefore, the reduction observed in the number of alarms due to repositioning can be higher if those techniques are more precisely implemented.

To compute efficiency obtained by our proposed techniques to improve AFPE efficiency (Chapters 5 and 6), we applied AFPE *without* and *with* the proposed techniques, and measured the difference in *clock time* as the efficiency improvement. The findings of the techniques vary based on the time taken by AFPE, which in turn varies based on the AFPE implementation used. For example, the techniques presented to improve efficiency of AFPE depend on the characteristics of the model checker used. Therefore, the findings can vary greatly depending on the model checker used. Ideally, to address this threat, evaluations should be based on multiple model checkers. However, this increases the analysis time multi-fold. To limit the amount of analysis time, we evaluated the techniques using CBMC, which is a commonly used model checker for AFPE [34, 152, 153, 174]. As another example, the time taken by AFPE would vary based on whether preprocessing techniques such as grouping of related assertions, loops abstraction, and program slicing, are used before making the model checking calls. To address the threat, we implemented preprocessing techniques which have been used in earlier studies related to improving AFPE efficiency (Section 5.5.2). In our implementation, we verified the groups of assertions sequentially: one at a time. The time taken would vary based on whether the groups of assertions are verified in parallel (and the number of threads used), machine configurations, etc.

The delta alarms selected to evaluate techniques proposed for postprocessing delta alarms would vary based on the implementation computing delta alarms (Section 6.5). Our implementation to compute delta alarms is based on the earlier implementation of *impact analysis-based technique* proposed by Chimdyalwar and Kumar [36]. The computation of impacted alarms and their classification depends on how precisely PDGs are computed and how precisely the mapping of the code is created. For instance, we used *diff* to create the mapping of the code from two subsequent versions. The mapping would be more precise if more advanced program differencing techniques, such as AST Diff [61], are used. A more precise mapping can help to identify equivalent changes and thus ignore some of the changed statements during the postprocessing. As a result, delta alarms generated on those applications would vary, and therefore the findings of the techniques that postprocess them.

7.3.2.3 Measurement of Manual Inspection Effort

Evaluating the method to manually inspect groups of common-POI alarms, requires measuring the time required to inspect the groups in two settings: without the method, and with the method (Section 5.5.1). Ideally, in the evaluation of the manual inspection method, groups of common-POI alarms should be inspected by multiple reviewers because the inspection varies depending on the reviewer's expertise, skill, and familiarity with the code. Since manual inspection is

a time consuming activity, reviewers require to spend considerable amount of time to inspect alarms. Therefore, manual inspection of common-POI alarms was performed solely by the author without involving additional reviewers. This might lead to subjective bias in the measurement of time required.

7.3.3 External Validity

Threats to external validity concern the extent to which the evaluation results generalize, beyond the sample used (the alarms selected in the evaluation), to the entire population. The entire population of alarms can only be obtained by analyzing all applications for all possible verification properties, and such an evaluation is not possible. To mitigate the generalization threat, we selected a large number of alarms generated for five different properties and on a variety of applications. The properties selected are such that they are supported by bug finding as well as code proving tools. The applications selected are a mix of open source and industry applications, and written in a common programming language C, which is typically used to implement safety-critical applications. Due to the common coding practices, we expect that the findings would not vary much if alarms to be processed are generated by some other static analysis tool on different applications and for other verification properties.

7.4 Future Work

This section describes several directions for future research.

In Chapter 2, we found that the approaches that have been proposed for postprocessing of alarms are complementary. Moreover, many of the techniques implementing the same approach are orthogonal. Thus, to further improve postprocessing of alarms, multiple techniques and approaches can be combined together. Studying feasibility of the different combinations possible, and investigating their advantages and disadvantages, would be a valuable piece of future research.

In Chapter 3, we presented repositioning technique to reduce the number of alarms. However, the reduction obtained is found to be limited. In Chapter 4, to improve the reduction, we classified the controlling conditions of alarms as *impacting* and *non-impacting*. Considering the effect of non-impacting conditions of alarms during repositioning allowed to further reduce the number of alarms. We expect that the non-impacting controlling conditions can be used to improve other techniques as well, e.g., to reduce size of code slices generated for the alarms, which in turn can help techniques that are based on the code slices. Evaluating usage of these non-impacting controlling conditions in other techniques can be a direction for future research. Moreover, we find that, many of the controlling conditions identified as impacting do not impact alarms. Therefore, to benefit from the non-impacting controlling conditions of alarms, designing techniques to precisely classify the controlling conditions of alarms into the two classes can be an interesting direction for future research.

In Chapter 5, we reduced redundancy in manual inspection of common-POI alarms generated on partitioned code, and AFPE applied to these alarms. We could reduce the redundancy by taking into account that the alarms are generated for the same POI but appearing in multiple partitions. We believe that, on similar lines techniques implementing other postprocessing approaches, such as ranking and pruning, can be improved when they are applied to common-POI alarms. Designing the new techniques for the improvement and evaluating them can be a direction for future research.

In Chapters 5 and 6, during evaluation of the reuse-based technique to reduce the time taken by AFPE, we found that the context expansion approach does not take into account the hierarchy and structure of the calling functions. Based on the hierarchy and structure of calling functions, we plan to reduce the number of model checking calls that arise due to applying context expansion approach. In another finding during applying those improved AFPE techniques, we observed that the technique to group related assertions groups related alarms which belong to the same function. Moreover, during context expansion, model checking calls for related alarms belonging to different functions can be combined when the calls are made for common calling functions. Based on this observation, as a future work, we plan to formulate and evaluate a technique to reduce the overall number of model checking calls.

Selection of Techniques Going beyond the studies immediately related to this thesis, we have identified several topics that should be explored in the future. As discussed in Section 7.2, multiple techniques need to be combined to obtain better results through postprocessing. The combinations of the techniques however will increase the analysis time multi-fold. Hence, reducing the analysis time is equally important. One approach to address this problem can be to *quickly predict* the improvement that a time-consuming technique can provide before the technique is applied. To this end, criteria need to be identified for those techniques to predict results on a given application. For example, related alarms can be computed before repositioning alarms: if no related alarms are present, application of repositioning technique can be skipped. Based on time availability and prediction results of the applicable techniques, a subset of techniques can be selected from a set of applicable techniques.

Furthermore, considering the advancement in machine-learning based techniques and their application to process alarms (Section 2.4.3.1), we believe that machine learning can be used to select a subset of techniques from a set of applicable techniques, which are more suitable on a given application compared to the other techniques. To this end, machine learning can be applied to learn the types (structures) of applications on which each of the techniques provides better results, and then to select the most suitable technique(s) for a given new application.

Tuning Programming for Precise Static Analysis The imprecision of ASATs is due to multiple reasons, e.g., unconstrained environment, usage of dynamic memory allocation, and presence of loops whose termination cannot be proved or whose upper bound cannot be identified statically. To reduce the number of alarms generated due to these reasons, we believe in educating developers to write the code in such a way that can improve precision of ASATs. For example, assertions (annotations) can be added in the code to help ASATs inferring the restricted range of values taken by the input variables. The loops can be written such that their bound can be determined statically by the tools. It would be interesting to conduct research in this direction, similar to *The Power of Ten* proposed by Gerard J. Holzmann [87].

Compile-time Static Analysis Existing studies about usage of ASATs [20, 92] indicate that ASATs are used only when their usage is enforced as a part of team/organizational work-policy. Moreover, the usage of tools is seen as a separate activity from the code development. To improve adoption of ASATs and benefit from them, we believe that integrating static analysis into *compilers* can help to improve adoption of ASATs. However, this would mandate that the efficiency of static analysis is comparable to compilers. Future research can be directed on how static analysis can be made a part of compilers, without compromising performance of the com-

pilers. One possible approach is to perform light-weight analysis by considering the evolutionary nature of the code, where only the changed and impacted code gets analyzed.

Bibliography

[1] Polyspace Code Prover. http://in.mathworks.com/products/polyspace-code-prover/. [Online: accessed 01-March-2020].

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Symposium on Principles of Programming Languages*, pages 14–25. ACM, 2003.

[3] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *International Computer Software and Applications Conference*, pages 343–350. IEEE, 2006.

[4] Enas A. Alikhashashneh, Rajeev R. Raje, and James H. Hill. Using machine learning techniques to classify and predict static code analysis tool warnings. In *International Conference on Computer Systems and Applications*, pages 1–8. IEEE, 2018.

[5] Frances E. Allen. Control flow analysis. In *Symposium on Compiler Optimization*, pages 1–19. ACM, 1970.

[6] Simon Allier, Nicolas Anquetil, Andre Hora, and Stephane Ducasse. A framework to compare alert ranking algorithms. In *Working Conference on Reverse Engineering*, pages 277–285. ACM, 2012.

[7] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. A survival analysis-based prioritization of code checker warning: A case study using PMD. In *International Conference on Big Data, Cloud Computing, and Data Science Engineering*, pages 69–83. Springer, 2019.

[8] Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE software*, 20(4):42–50, 2003.

[9] Satoshi Arai, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. A gamified tool for motivating developers to remove warnings of bug pattern tools. In *International Workshop on Empirical Software Engineering in Practice*, pages 37–42. IEEE, 2014.

[10] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *International Workshop on State Of the Art in Program Analysis*, pages 1–6. ACM, 2015.

[11] Nathaniel Ayewah and William Pugh. Using checklists to review static analysis warnings. In *International Workshop on Defects in Large Software Systems*, pages 11–15. ACM, 2009.

[12] Nathaniel Ayewah and William Pugh. The Google FindBugs fixit. In *International Symposium on Software Testing and Analysis*, pages 241–252. ACM, 2010.

[13] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[14] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 1–8. ACM, 2007.

[15] Deepika Badampudi, Claes Wohlin, and Kai Petersen. Experiences from using snowballing and database searches in systematic literature studies. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 17:1–17:10. ACM, 2015.

[16] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[17] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. In *International Static Analysis Symposium*, pages 337–354. Springer, 2003.

[18] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *International Conference on Software Maintenance and Evolution*, pages 211–221. IEEE, 2016.

[19] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 613–624. ACM, 2019.

[20] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 470–481. IEEE, 2016.

[21] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[22] Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Conference on Programming Language Design and Implementation*, pages 209–218. ACM, 2013.

[23] Cathal Boogerd and Leon Moonen. Prioritizing software inspection results using static profiling. In *International Workshop on Source Code Analysis and Manipulation*, pages 149–160. IEEE, 2006.

[24] Guillaume Brat and Arnaud Venet. Precise and scalable static program analysis of NASA flight software. In *Aerospace Conference*, pages 1–10. IEEE, 2005.

[25] Guillaume Brat and Willem Visser. Combining static analysis and model checking for software analysis. In *International Conference on Automated Software Engineering*, pages 262–269. IEEE, 2001.

[26] Tim Buckers, Clinton Cao, Michiel Doesburg, Boning Gong, Sunwei Wang, Moritz Beller, and Andy Zaidman. UAV: Warnings from multiple automated static analysis tools at a glance. In *International Conference on Software Analysis, Evolution and Reengineering*, pages 472–476. IEEE, 2017.

[27] Aji Ery Burhandenny, Hirohisa Aman, and Minoru Kawahara. Investigation of coding violations focusing on authorships of source files. In *International Conference on Applied Computing and Information Technology/International Conference on Computational Science Intelligence and Applied Informatics/International Conference on Big Data, Cloud Computing, Data Science*, pages 248–253. IEEE, 2017.

[28] CBMC. `http://www.cprover.org/cbmc/`. [Online: accessed 01-March-2020].

[29] Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. Learning a strategy for choosing widening thresholds from a large codebase. In *Asian Symposium on Programming Languages and Systems*, pages 25–41. Springer, 2016.

[30] George Chatzieleftheriou and Panagiotis Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. In *Computer Software and Applications Conference Workshops*, pages 96–103. IEEE, 2011.

[31] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *Symposium on Applied Computing*, pages 1284–1291. ACM, 2012.

[32] Chen Chen, Kai Lu, Xiaoping Wang, Xu Zhou, and Li Fang. Pruning false positives of static data-race detection via thread specialization. In *International Workshop on Advanced Parallel Processing Technologies*, pages 77–90. Springer, 2013.

[33] Ping Chen, Hao Han, Yi Wang, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. IntFinder: Automatically detecting integer bugs in x86 binary program. In *International Conference on Information and Communications Security*, pages 336–345. Springer, 2009.

[34] Bharti Chimdyalwar and Priyanka Darke. Statically relating program properties for efficient verification (short WIP paper). In *International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 99–103. ACM, 2018.

[35] Bharti Chimdyalwar, Priyanka Darke, Anooj Chavda, Sagar Vaghani, and Avriti Chauhan. Eliminating static analysis false positives using loop abstraction and bounded model checking. In *International Symposium on Formal Methods*, pages 573–576. Springer, 2015.

[36] Bharti Chimdyalwar and Shrawan Kumar. Effective false positive filtering for evolving software. In *India Software Engineering Conference*, pages 103–106. ACM, 2011.

[37] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *International Conference on Automated Software Engineering*, pages 332–343. ACM, 2016.

[38] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[39] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[40] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[41] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: Combining static checking and testing. In *International Conference on Software Engineering*, pages 422–431. ACM, 2005.

[42] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–37, 2008.

[43] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C. In *International Conference on Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[44] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[45] Priyanka Darke, Bharti Chimdyalwar, Avriti Chauhan, and R. Venkatesh. Efficient safety proofs for industry-scale code using abstractions and bounded model checking. In *International Conference on Software Testing, Verification and Validation*, pages 468–475. IEEE, 2017.

[46] Priyanka Darke, Bharti Chimdyalwar, R. Venkatesh, Ulka Shrotri, and Ravindra Metta. Over-approximating loops to prove properties using bounded model checking. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1407–1412. IEEE, 2015.

[47] Priyanka Darke, Mayur Khanzode, Arun Nair, Ulka Shrotri, and R. Venkatesh. Precise analysis of large industry code. In *Asia-Pacific Software Engineering Conference*, pages 306–309. IEEE, 2012.

[48] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *International Conference on Computer Aided Verification*, pages 324–342. Springer, 2015.

[49] Vinicius Rafael Lobo de Mendonca, Cassio Leonardo Rodrigues, Fabrízzio Alphonsus Alves de Melo Nunes Soares, and Auri Marcelo Rizzo Vincenzi. Static analysis techniques and tools: A systematic mapping study. In *International Conference on Software Engineering Advances*, pages 72–78. IARIA XPS Press, 2013.

[50] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *International Conference on Computer Aided Verification*, pages 20–36. Springer, 2007.

[51] David Delmas and Jean Souyris. Astrée: From research to industry. In *International Static Analysis Symposium*, pages 437–451. Springer, 2007.

[52] Ewen Denney and Steven Trac. A software safety certification tool for automatically generated guidance, navigation and control code. In *Aerospace Conference*, pages 1–11. IEEE, 2008.

[53] Edsger W. Dijkstra. Structured programming. In *Classics in Software Engineering*, pages 41–48. Yourdon Press, 1979.

[54] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In *Conference on Programming Language Design and Implementation*, pages 181–192. ACM, 2012.

[55] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *International Symposium on Software Testing and Analysis*, pages 307–317. ACM, 2017.

[56] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[57] Matthew B. Dwyer, John Hatcliff, Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *Future of Software Engineering*, pages 120–136. IEEE, 2007.

[58] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Information and Software Technology*, 54(1):1–15, 2012.

[59] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008.

[60] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 191–210. Springer, 2004.

[61] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, pages 313–324. ACM, 2014.

[62] Ansgar Fehnker and Ralf Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Innovations in Systems and Software Engineering*, 9(1):45–56, 2013.

[63] Josselin Feist, Laurent Mounier, Sébastien Bardin, Robin David, and Marie-Laure Potet. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Workshop on Software Security, Protection, and Reverse Engineering*, page 2. ACM, 2016.

[64] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[65] Jean-Christophe Filliâtre. Deductive software verification. Springer, 2011.

[66] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. Prioritizing alerts from multiple static analysis tools, using classification models. In *International Workshop on Software Qualities and Their Dependencies*, pages 13–20. ACM, 2018.

[67] Zachary P. Fry and Westley Weimer. Clustering static analysis defect reports to reduce maintenance costs. In *Working Conference on Reverse Engineering*, pages 282–291. IEEE, 2013.

[68] Mikhail R. Gadelha, Enrico Steffinlongo, Lucas C. Cordeiro, Bernd Fischer, and Denis Nicole. SMT-based refutation of spurious bug reports in the Clang Static Analyzer. In *International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 11–14. IEEE, 2019.

[69] Fengjuan Gao, Linzhang Wang, and Xuandong Li. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In *International Conference on Automated Software Engineering*, pages 786–791. IEEE, 2016.

[70] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: dynamic symbolic execution guided with static verification results. In *International Conference on Software Engineering*, pages 992–994. ACM, 2011.

[71] Marcel Gehrke. Bidirectional predicate propagation in Frama-C and its application to warning removal. Master's thesis, Hamburg University of Technology, 2014.

[72] Alexander Yu. Gerasimov. Directed dynamic symbolic execution for static analysis warnings confirmation. *Programming and Computer Software*, 44(5):316–323, Sep 2018. Translated by O. Pismenov.

[73] Alexander Yu. Gerasimov and Leonid V. Kruglov. Reachability confirmation of statically detected defects using dynamic analysis. In *Ivannikov Memorial Workshop*, pages 24–31. IEEE, 2018.

[74] Alexander Yu. Gerasimov, Leonid V. Kruglov, Mikhail K. Ermakov, and Sergey P. Vartanov. An approach to reachability determination for static analysis defects with the help of dynamic symbolic execution. In *Programming and Computer Software*, volume 44, pages 467–475. Springer, 2018.

[75] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. Automatically refining abstract interpretations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–458. Springer, 2008.

[76] Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *International Symposium on Software Testing and Analysis*, pages 318–328. ACM, 2017.

[77] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Working Conference on Mining Software Repositories*, pages 152–161. ACM, 2014.

[78] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization*, pages 289–298. IEEE, 2011.

[79] Mark Harman, Malcolm Munro, Lin Hu, and Xingyuan Zhang. Side-effect removal transformation. In *International Workshop on Program Comprehension*, pages 310–319. IEEE, 2001.

[80] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *International Working Conference on Source Code Analysis and Manipulation*, pages 1–23. IEEE, 2018.

[81] Sarah Heckman. Adaptively ranking alerts generated from automated static analysis. *XRDS: Crossroads, The ACM Magazine for Students*, 14(1):1–11, 2007.

[82] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *International Symposium on Empirical Software Engineering and Measurement*, pages 41–50. ACM, 2008.

[83] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *International Conference on Software Testing Verification and Validation*, pages 161–170. IEEE, 2009.

[84] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, 2011.

[85] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *International Conference on Software Engineering*, pages 519–529. IEEE, 2017.

[86] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. Continuously reasoning about programs using differential Bayesian inference. In *Conference on Programming Language Design and Implementation*, pages 561–575. ACM, 2019.

[87] Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[88] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[89] Pankaj Jalote, Vipindeep Vangala, Taranbir Singh, and Prateek Jain. Program partitioning: A framework for combining static and dynamic analysis. In *International Workshop on Dynamic Systems Analysis*, pages 11–16. ACM, 2006.

[90] Raoul Praful Jetley, Paul L. Jones, and Paul Anderson. Static analysis of medical device software using CodeSonar. In *Workshop on Static Analysis*, pages 22–29. ACM, 2008.

[91] Jun-li Jiao, Da-hai Jin, and Ming-nan Zhou. Dominant alarm research and implementation based on static defect detection. *DEStech Transactions on Computer Science and Engineering*, (cimns), 2017.

[92] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering*, pages 672–681. IEEE, 2013.

[93] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. *ACM SIGPLAN Notices*, 47(1):19–30, 2012.

[94] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *International Static Analysis Symposium*, pages 203–217. Springer, 2005.

[95] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. SMT-based false positive elimination in static program analysis. In *International Conference on Formal Engineering Methods*, pages 316–331. Springer, 2012.

[96] Bishoksan Kafle, John P. Gallagher, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. An iterative approach to precondition inference using constrained Horn clauses. *Theory and Practice of Logic Programming*, 18(3-4):553–570, 2018.

[97] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[98] Jasper Kamperman. Automated software inspection: A new approach to increased software quality and productivity. *Reasoning Inc., White paper*, 2002.

[99] Shubhangi Khare, Sandeep Saraswat, and Shrawan Kumar. Static program analysis of large embedded code base: An experience. In *India Software Engineering Conference*, pages 99–102. ACM, 2011.

[100] Uday P. Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, 2017.

[101] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Path projection for user-centered static analysis tools. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 57–63. ACM, 2008.

[102] Joon-Ho Kim, Myung-Chul Ma, and Jae-Pyo Park. An analysis on secure coding using symbolic execution engine. *Journal of Computer Virology and Hacking Techniques*, 12(3):177–184, 2016.

[103] Sunghun Kim and Michael D. Ernst. Prioritizing warning categories by analyzing software history. In *International Workshop on Mining Software Repositories*, pages 27–27. IEEE, 2007.

[104] Sunghun Kim and Michael D. Ernst. Which warnings should I fix first? In *Joint meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 45–54. ACM, 2007.

[105] Youil Kim, Jooyong Lee, Hwansoo Han, and Kwang-Moo Choe. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology*, 52(2):210–219, 2010.

[106] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[107] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. Combining static and dynamic analyses for vulnerability detection: illustration on heartbleed. In *Haifa Verification Conference*, pages 39–50. Springer, 2015.

[108] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2007.

[109] John Knight. Safety critical systems: challenges and directions. In *International Conference on Software Engineering*, pages 547–550. IEEE, 2002.

[110] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. Learning a classifier for false positive error reports emitted by static code analysis tools. In *International Workshop on Machine Learning and Programming Languages*, pages 35–42. ACM, 2017.

[111] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. In *International Conference on Scalable Information Systems*, pages 55:1–55:7. ICST, 2007.

[112] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5(2):149–161, 2009.

[113] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *International Symposium on Foundations of Software Engineering*, pages 83–93. ACM, 2004.

[114] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *International Static Analysis Symposium*, pages 295–315. Springer, 2003.

[115] Rahul Kumar and Aditya V. Nori. The economics of static analysis tools. In *Joint Meeting on Foundations of Software Engineering*, pages 707–710. ACM, 2013.

[116] Shrawan Kumar, Amitabha Sanyal, and Uday P. Khedker. Value slice: A new slicing concept for scalable property checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 101–115. Springer, 2015.

[117] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. Differential assertion checking. In *Joint Meeting on Foundations of Software Engineering*, pages 345–355. ACM, 2013.

[118] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[119] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process*, 28(7):589–618, 2016.

[120] Lucas Layman, Laurie Williams, and Robert St. Amant. 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*, pages 176–185. IEEE, 2007.

[121] Wei Le and Mary Lou Soffa. Path-based fault correlations. In *International Symposium on Foundations of Software Engineering*, pages 307–316. ACM, 2010.

[122] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *Conference on Software Testing, Validation and Verification*, pages 391–401. IEEE, 2019.

[123] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. *ACM Transactions on Programming Languages and Systems*, 39(4):1–35, 2017.

[124] Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 299–314. Springer, 2012.

[125] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *International Conference on Availability, Reliability and Security*, pages 446–454. IEEE, 2013.

[126] J. Jenny Li, John Palframan, and Jim Landwehr. SoftWare IMmunization (SWIM) - a combination of static analysis and automatic testing. In *Annual Computer Software and Applications Conference*, pages 656–661. IEEE, 2011.

[127] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. Residual investigation: Predictive and precise bug detection. In *International Symposium on Software Testing and Analysis*, pages 298–308. ACM, 2012.

[128] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of Android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[129] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. Dynamically validating static memory leak warnings. In *International Symposium on Software Testing and Analysis*, pages 112–122. ACM, 2013.

[130] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. Automatic construction of an effective training set for prioritizing static analysis warnings. In *International Conference on Automated Software Engineering*, pages 93–102. ACM, 2010.

[131] Guangtai Liang, Qian Wu, Qianxiang Wang, and Hong Mei. An effective defect detection and warning prioritization approach for resource leaks. In *Annual Computer Software and Applications Conference*, pages 119–128. IEEE, 2012.

[132] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for FindBugs violations. *IEEE Transactions on Software Engineering*, 2018.

[133] Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *Conference on Programming Language Design and Implementation*, pages 294–304. ACM, 2014.

[134] Alex Loh and Miryung Kim. LSdiff: A program differencing tool to identify systematic structural differences. In *International Conference on Software Engineering*, pages 263–266. ACM, 2010.

[135] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *International Symposium on Code Generation and Optimization*, pages 136–146. IEEE, 2009.

[136] Bailin Lu, Wei Dong, Liangze Yin, and Li Zhang. Evaluating and integrating diverse bug finders for effective program analysis. In *International Conference on Software Analysis, Testing, and Evolution*, pages 51–67. Springer, 2018.

[137] Xutong Ma, Jiwei Yan, Jun Yan, and Jian Zhang. Reorganizing and optimizing post-inspection on suspicious bug reports in path-sensitive analysis. In *International Conference on Software Quality, Reliability and Security*, pages 260–271. IEEE, 2019.

[138] Stephen MacDonell, Martin Shepperd, Barbara Kitchenham, and Emilia Mendes. How reliable are systematic reviews in empirical software engineering? *IEEE Transactions on Software Engineering*, 36(5):676–687, 2010.

[139] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In *Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[140] Diego Marcilio, Carlo A. Furia, Rodrigo Bonifácio, and Gustavo Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *International Working Conference on Source Code Analysis and Manipulation*, pages 34–44. IEEE, 2019.

[141] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *International Conference on World Wide Web*, pages 63–74. ACM, 2014.

[142] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. An approach to merge results of multiple static analysis tools (short paper). In *International conference on quality software*, pages 169–174. IEEE, 2008.

[143] Qingkun Meng, Chao Feng, Bin Zhang, and Chaojing Tang. Assisting in auditing of buffer overflow vulnerabilities via machine learning. *Mathematical Problems in Engineering*, 2017, 2017.

[144] Maxim Menshchikov and Timur Lepikhin. 5w+ 1h static analysis report quality measure. In *International Conference on Tools and Methods for Program Analysis*, pages 114–126. Springer, 2017.

[145] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[146] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Symposium on Program as Data Objects*, pages 155–172. Springer, 2001.

[147] David Monniaux. A survey of satisfiability modulo theory. In *International Workshop on Computer Algebra in Scientific Computing*, pages 401–425. Springer, 2016.

[148] Tukaram Muske. Improving review of clustered-code analysis warnings. In *International Conference on Software Maintenance and Evolution*, pages 569–572. IEEE, 2014.

[149] Tukaram Muske. Supporting reviewing of warnings in presence of shared variables: Need and effectiveness. In *International Symposium on Software Reliability Engineering Workshops*, pages 104–107. Springer, 2014.

[150] Tukaram Muske, Ankit Baid, and Tushar Sanas. Review efforts reduction by partitioning of static analysis warnings. In *International Working Conference on Source Code Analysis and Manipulation*, pages 106–115. IEEE, 2013.

[151] Tukaram Muske and Prasad Bokil. On implementational variations in static analysis tools. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 512–515. IEEE, 2015.

[152] Tukaram Muske, Advaita Datar, Mayur Khanzode, and Kumar Madhukar. Efficient elimination of false positives using bounded model checking. In *International Conference on Advances in System Testing and Validation Lifecycle*, pages 13–20. IARIA XPS Press, 2013.

[153] Tukaram Muske and Uday P. Khedker. Efficient elimination of false positives using static analysis. In *International Symposium on Software Reliability Engineering*, pages 270–280. IEEE, 2015.

[154] Tukaram Muske and Uday P. Khedker. Cause points analysis for effective handling of alarms. In *International Symposium on Software Reliability Engineering*, pages 173–184. IEEE, 2016.

[155] Tukaram Muske and Alexander Serebrenik. Survey of approaches for handling static analysis alarms. In *International Working Conference on Source Code Analysis and Manipulation*, pages 157–166. IEEE, 2016.

[156] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of static analysis alarms. In *International Symposium on Software Testing and Analysis*, pages 187–197. ACM, 2018.

[157] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Reducing static analysis alarms based on non-impacting control dependencies. In *Asian Symposium on Programming Languages and Systems*, pages 115–135. Springer, 2019.

[158] Tukaram Muske and Amey Zare. Inconsistencies-based multi-region protocol verification. In *International Conference on Advances in System Testing and Validation Lifecycle*, pages 40–45. IARIA XPS Press, 2014.

[159] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *International Conference on Software Engineering*, pages 580–586. ACM, 2005.

[160] Thu-Trang Nguyen, Toshiaki Aoki, Takashi Tomita, and Iori Yamada. Multiple program analysis techniques enable precise check for SEI CERT C coding standard. In *Asia-Pacific Software Engineering Conference*, pages 70–77. IEEE, 2019.

[161] Thu Trang Nguyen, Pattaravut Maleehuan, Toshiaki Aoki, Takashi Tomita, and Iori Yamada. Reducing false positives of static analysis for SEI CERT C coding standard. In *Joint International Workshop on Conducting Empirical Studies in Industry and International Workshop on Software Engineering Research and Industrial Practice*, pages 41–48. IEEE, 2019.

[162] Lisa Nguyen Quang Do and Eric Bodden. Gamifying static analysis. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 714–718. ACM, 2018.

[163] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[164] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*, 101(2):161–185, 2019.

[165] Hideto Ogasawara, Minoru Aizawa, and Atsushi Yamada. Experiences with program static analysis. In *International Software Metrics Symposium*, pages 109–112. IEEE, 1998.

[166] Jan-Peter Ostberg and Stefan Wagner. At ease with your warnings: The principles of the salutogenesis model applied to automatic static analysis. In *International Conference on Software Analysis, Evolution, and Reengineering*, pages 629–633. IEEE, 2016.

[167] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis. *IET Software*, 10:54–61, 2016.

[168] Chris Parnin, Carsten Görg, and Ogechi Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Symposium on Software Visualization*, pages 77–86. ACM, 2008.

[169] Riyad Parvez, Paul A. S. Ward, and Vijay Ganesh. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries. In *International Conference on Computer Science and Software Engineering*, pages 116–127. IBM Corp., 2016.

[170] Étienne Payet and Fausto Spoto. Static analysis of Android programs. *Information and Software Technology*, 54(11):1192–1201, 2012.

[171] Khoo Yit Phang, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. Triaging checklists: a substitute for a PhD in static analysis. *Evaluation and Usability of Programming Languages and Tools*, 2009, 2009.

[172] Marco Pistoia, Omer Tripp, and David Lubensky. Combining static code analysis and machine learning for automatic detection of security vulnerabilities in mobile apps. In *Mobile Application Development, Usability, and Security*, pages 68–94. IGI Global, 2017.

[173] Andreas Podelski, Martin Schäf, and Thomas Wies. Classifying bugs with interpolants. In *International Conference on Tests and Proofs*, pages 151–168. Springer, 2016.

[174] Hendrik Post, Carsten Sinz, Alexander Kaiser, and Thomas Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *International Conference on Automated Software Engineering*, pages 188–197. IEEE, 2008.

[175] Louis-Philippe Querel and Peter C. Rigby. WarningsGuru: integrating statistical bug models with static analysis to provide timely and specific bug warnings. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 892–895. ACM, 2018.

[176] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using Bayesian inference. In *Conference on Programming Language Design and Implementation*, pages 722–735. ACM, 2018.

[177] I.V. Ramakrishnan, Prasad Rao, Konstantinos Sagonas, Terrance Swift, and David S. Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711. The MIT Press, 1995.

[178] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.

[179] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking source code static analysis warnings for continuous monitoring of FLOSS repositories. In *IFIP International Conference on Open Source Systems*, pages 90–101. Springer, 2018.

[180] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. Ranking warnings from multiple source code static analyzers via ensemble learning. In *International Symposium on Open Collaboration*, pages 1–10. ACM, 2019.

[181] Xavier Rival. Abstract dependences for alarm diagnosis. In *Asian Symposium on Programming Languages and Systems*, pages 347–363. Springer, 2005.

[182] Xavier Rival. Understanding the origin of alarms in ASTRÉE. In *International Static Analysis Symposium*, pages 303–319. Springer, 2005.

[183] Neha Rungta and Eric G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *Haifa Verification Conference*, pages 23–37. Springer, 2008.

[184] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *International Symposium on Software Reliability Engineering*, pages 245–256. IEEE, 2004.

[185] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *International conference on Software engineering*, pages 341–350. ACM, 2008.

[186] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering*, pages 598–608. IEEE, 2015.

[187] Haihao Shen, Jianhong Fang, and Jianjun Zhao. EFindbugs: Effective error ranking for FindBugs. In *International Conference on Software Testing, Verification and Validation*, pages 299–308. IEEE, 2011.

[188] Mark S. Sherriff, Sarah Heckman, J. Michael Lake, and Laurie Williams. Using groupings of static analysis alerts to identify files likely to contain field failures. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering: Companion Papers*, pages 565–568. ACM, 2007.

[189] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3):1–41, 2012.

[190] Hasan Sözer. Integrated static code analysis and runtime verification. *Software: Practice and Experience*, 45(10):1359–1373, 2015.

[191] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking defect warnings across versions. In *International workshop on Mining software repositories*, pages 133–136. ACM, 2006.

[192] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Conference on Programming Language Design and Implementation*, pages 112–122. ACM, 2007.

[193] Y. N. Srikant and Priti Shankar. *The compiler design handbook: optimizations and machine code generation*. CRC Press, 2007.

[194] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: A critical review and guidelines. In *International Conference on Software Engineering*, pages 120–131. ACM, 2016.

[195] David Svoboda, Lori Flynn, and Will Snavely. Static analysis alert audits: Lexicon & rules. In *Cybersecurity Development*, pages 37–44. IEEE, 2016.

[196] Terrance Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.

[197] TCS Embedded Code Analyzer (TCS ECA). `https://www.tcs.com/tcs-embedded-code-analyzer`. [Online: accessed 01-March-2020].

[198] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *International Conference on Automated Software Engineering*, pages 50–59. IEEE, 2012.

[199] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[200] Vassil Todorov, Frédéric Boulanger, and Safouan Taha. Formal verification of automotive embedded software. In *Conference on Formal Methods in Software Engineering*, pages 84–87. ACM, 2018.

[201] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *International Symposium on Software Testing and Analysis*, pages 97–107. ACM, 2007.

[202] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. ALETHEIA: Improving the usability of static security analysis. In *Conference on Computer and Communications Security*, pages 762–774. ACM, 2014.

[203] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[204] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *Conference on Programming Language Design and Implementation*, pages 87–97. ACM, 2009.

[205] Manuel Valdiviezo, Cristina Cifuentes, and Padmanabhan Krishnan. A method for scalable and precise bug finding using program analysis and model checking. In *Asian Symposium on Programming Languages and Systems*, pages 196–215. Springer, 2014.

[206] Arnaud Venet. A practical approach to formal software verification by static analysis. *ACM SIGAda Ada Letters*, 28(1):92–95, 2008.

[207] Radhika D. Venkatasubramanyam and Shrinath Gupta. An automated approach to detect violations with high confidence in incremental code using a learning system. In *International Conference on Software Engineering (Companion Proceedings)*, pages 472–475. ACM, 2014.

[208] Han Wang, Min Zhou, Xi Cheng, Guang Chen, and Ming Gu. Which defect should be fixed first? semantic prioritization of static analysis report. In *International Conference on Software Analysis, Testing, and Evolution*, pages 3–19. Springer, 2018.

[209] Lei Wang, Qiang Zhang, and PengChao Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *International Working Conference on Source Code Analysis and Manipulation*, pages 165–173. IEEE, 2008.

[210] Lili Wei, Yepang Liu, and Shing-Chi Cheung. OASIS: Prioritizing static analysis warnings for Android apps based on app user reviews. In *Joint Meeting on Foundations of Software Engineering*, pages 672–682. ACM, 2017.

[211] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449. IEEE, 1981.

[212] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[213] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *International Conference on Evaluation and Assessment in Software Engineering*, pages 38:1–38:10. ACM, 2014.

[214] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity concerns in software engineering research. In *Workshop on Future of software engineering research*, pages 411–414. ACM, 2010.

[215] Mingjie Xu, Shengnan Li, Lili Xu, Feng Li, Wei Huo, Jing Ma, Xinhua Li, and Qingjia Huang. A light-weight and accurate method of static integer-overflow-to-buffer-overflow vulnerability detection. In *International Conference on Information Security and Cryptology*, pages 404–423. Springer, 2018.

[216] Jiangtao Xue, Xinjun Mao, Yao Lu, Yue Yu, and Shangwen Wang. History-driven fix for code quality issues. *IEEE Access*, 7:111637–111648, 2019.

[217] Achilleas Xypolytos, Haiyun Xu, Barbara Vieira, and Amr M. T. Ali-Eldin. A framework for combining and ranking static analysis tool findings based on tool performance statistics. In *International Conference on Software Quality, Reliability and Security Companion*, pages 595–596. IEEE, 2017.

[218] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Information Processing Letters*, 102(2-3):118–123, 2007.

[219] Jongwon Yoon, Minsik Jin, and Yungbum Jung. Reducing false alarms from an industrial-strength static analyzer by SVM. In *Asia-Pacific Software Engineering Conference*, volume 2, pages 3–6. IEEE, 2014.

[220] Lian Yu, Jun Zhou, Yue Yi, Jianchu Fan, and Qianxiang Wang. A hybrid approach to detecting security defects in programs. In *International Conference on Quality Software*, pages 1–10. IEEE, 2009.

[221] Ulas Yüksel and Hasan Sözer. Automated classification of static code analysis alerts: a case study. In *International Conference on Software Maintenance*, pages 532–535. IEEE, 2013.

[222] Bin Zhang, Chao Feng, Bo Wu, and Chaojing Tang. Detecting integer overflow in windows binary executables based on symbolic execution. In *International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 385–390. IEEE, 2016.

[223] Dalin Zhang, Dahai Jin, Yunzhan Gong, and Hailong Zhang. Diagnosis-oriented alarm correlations. In *Asia-Pacific Software Engineering Conference*, pages 172–179. IEEE, 2013.

[224] Dalin Zhang, Dahai Jin, Ying Xing, Hailong Zhang, and Yunzhan Gong. Automatically mining similar warnings and warning combinations. In *International Conference on Fuzzy Systems and Knowledge Discovery*, pages 783–788. IEEE, 2013.

[225] Danfeng Zhang and Andrew C. Myers. Toward general diagnosis of static errors. In *Symposium on Principles of Programming Languages*, pages 569–581. ACM, 2014.

[226] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[227] Yuwei Zhang, Ying Xing, Yunzhan Gong, Dahai Jin, Honghui Li, and Feng Liu. A variable-level automated defect identification model based on machine learning. *Soft Computing*, 24(2):1045–1061, 2020.

[228] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, and Mladen A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

[229] Zeineb Zhioua, Stuart Short, and Yves Roudier. Static code analysis for software security verification: Problems and approaches. In *International Computer Software and Applications Conference Workshops*, pages 102–109. IEEE, 2014.

[230] Honglei Zhu, Dahai Jin, and Yunzhan Gong. Inter-procedural diagnosis path generation for automatic confirmation of program suspected faults. *Tehnički vjesnik*, 26(3):762–770, 2019.

Curriculum Vitae

Tukaram Bhagwat Muske was born on July 11, 1984, in Hipparsoga, Maharashtra, India. He obtained his Bachelor of Engineering degree in Computer Science in 2006 at Maharashtra Institute of Technology, affiliated to University of Pune, India. After obtaining his Bachelor's degree, he worked with IBM India as a developer. In 2007 he joined Tata Research Development and Design Centre (TRDDC), a research wing of Tata Consultancy Services Limited, located in Pune. Since then he has been working at TRDDC, first as a developer and later as a scientist. Initially, he worked on designing a technique for structuring and composing of software development artifacts in software product line engineering, and then on developing tools to automatically generate test data for different coverage criteria.

Between 2011 and 2013, he was involved in designing and development of a commercial tool, TCS Embedded Code Analyzer (TCS ECA). He contributed to design and implementation of techniques for detecting defects of unique types, and techniques for improving precision of underlying analyses such as value and pointer analysis. Since 2013 he has been working as scientist, with the focus on addressing the problem of static analysis alarms generated by static analysis tools. To address the problem, he has proposed several techniques to improve postprocessing of alarms. Based on his development and research work at TRDDC, he has (co)-authored 14 peer-reviewed conference papers, and filed 12 unique patents of which seven are granted and the other five are currently under examination.

Since 2016 he has also been pursuing his PhD degree under supervision of Dr. Alexander Serebrenik and Prof. Dr. M.G.J. van den Brand, Eindhoven University of Technology, The Netherlands. He has gained research experience and expertise in static analysis, software engineering, applications of model checking, and verification and validation of embedded systems.

ISBN: 978-90-386-5037-1