Techniques for Efficient Automated Elimination of False Positives

Tukaram Muske
Tata Research Development and Design Centre
Tata Consultancy Services, Pune, India
t.muske@tcs.com

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

***Abstract*—Static analysis tools are useful to detect common programming errors. However, they generate a large number of false positives. Postprocessing of these alarms using a model checker has been proposed to automatically eliminate false positives from them. To scale up the *automated false positives elimination* (AFPE), several techniques, e.g., *program slicing*, are used. However, these techniques increase the time taken by AFPE, and the increased time is a major concern during application of AFPE to alarms generated on large systems.**

To reduce the time taken by AFPE, we propose two techniques. The techniques achieve the reduction by identifying and skipping redundant calls to the slicer and model checker. The first technique is based on our observation that, (a) combination of application-level slicing, verification with incremental context, and the context-level slicing helps to eliminate more false positives; (b) however, doing so can result in redundant calls to the slicer. In this technique, we use data dependencies to compute these redundant calls. The second technique is based on our observation that (a) *code partitioning* is commonly used by static analysis tools to analyze very large systems, and (b) applying AFPE to alarms generated on partitioned-code can result in repeated calls to both the slicer and model checker. We use memoization to identify the repeated calls and skip them.

The first technique is currently under evaluation. Our initial evaluation of the second technique indicates that it reduces AFPE time by up to 56%, with median reduction of 12.15%.

***Index Terms*—Static analysis, alarms/warnings, false positives, model checking, program slicing, code partitioning**

I. Introduction

Static analysis tools help to detect common programming errors in software systems and even to ensure that the systems are free of such defects [1], [2]. Despite this, recent studies [3]–[5] report that these tools are underused in practice. The studies report that the large number of false positives generated is a primary reason for the underuse: in general there are 40 alarms for every thousand lines of code [5], and 35% to 91% of alarms are false positives [6]. Partitioning alarms into false positives and errors requires manual inspection which is tedious, and time-consuming [3]. Moreover, the manual inspection of alarms even can be error-prone [7].

To reduce the number of false positives reported and thus the cost of manual inspection, postprocessing of alarms using a model checker has been proposed [8], [9]. In this postprocessing, (a) an assertion is generated corresponding to each alarm such that the assertion holds if and only if the alarm is a false positive; (b) the assertion is verified using a model checker; and (c) the alarm is eliminated as a false positive if the verification concludes that the assertion holds.

Due to the state space explosion problem, model checkers are known to fail to verify large systems [10]. To address this problem during *automated false positives elimination* (AFPE), several techniques have been proposed. These techniques include *verification using incremental context expansion* (that we call *VerICE*) [10], *context-level slicing* (slicing the code before each model checking call in its verification context) [8], [11], and *grouping of related assertions* [12]. Although the combination of these techniques helps to scale up AFPE to large systems, it considerably increases the number of calls made to the slicer and model checker (more details in Section II). Prior evaluations of AFPE in industry setting indicate that processing a group of related assertions, on average, results in five calls to both the slicer and model checker [8], [11], [12].

Due to 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: on average, processing a group of related alarms takes around three to four minutes [8], [11], [12]. This ultimately renders AFPE unsuitable to postprocess alarms generated on large systems. Therefore, improving efficiency of AFPE, i.e., reducing the time taken by AFPE, is important.

To improve efficiency of AFPE, we propose two automatic techniques. We achieve the improvement by identifying redundant calls to the slicer and model checker: skipping those calls does not affect the false positives eliminated. We stress that the identification of redundant calls by the techniques proposed is correct but not necessarily complete, i.e., all calls identified as redundant are redundant, but there might be redundant calls not identified as such by the techniques proposed. We design the two techniques based on our observations below.

1) Technique 1: We find that, while AFPE with the combination of application-level slicing, VerICE, and context-level slicing helps to eliminate false positives, it results in redundant calls to the slicer. To compute and skip the redundant calls, we design *a novel technique based on data dependencies*.

2) Technique 2: Code partitioning, i.e., breaking the system into smaller partitions and analyzing them separately, is commonly used by static analysis tools to analyze very large systems [13]–[15]. It is common that two or more partitions formed overlap: they share some functionality and the code

(a set of functions) that implements the functionality is also a part of each of those partitions. During the partition-wise static analysis, for an expression appearing in a function common to two or more partitions, multiple alarms of same property can get generated (Section IV-A). We find that, applying AFPE to such alarms results in repeated calls to both the slicer and model checker. Based on this observation, we *design a technique that uses memoization to skip those repeated calls*.

The first technique is currently under evaluation. Our initial evaluation of the second technique indicates that, the technique identifies up to 58.5% of the model checking calls with their corresponding slicing calls as redundant, while the median being 19.8%. Skipping these redundant calls reduces AFPE time by up to 56%, with median reduction of 12.15%.

Following are the key contributions of this paper.

1) A technique to improve efficiency of AFPE by eliminating redundant calls to the slicer.
2) A technique to improve efficiency of AFPE applied to alarms generated on partitioned-code.

Outline: Section II briefly discusses the techniques used to scale AFPE on large systems. Sections III and IV describe the two techniques proposed to improve AFPE efficiency. Section V presents our initial evaluations. Section VI discusses related work, and Section VII presents conclusions and future work.

II. Background & Notations

Following we describe existing techniques proposed for scalability of AFPE. We begin by presenting examples of alarms, and notations, that we use to describe the techniques.

A. Alarm Examples and Notations

Consider the code example shown in Figure 1, crafted to illustrate need of combination of multiple techniques for scalable AFPE. The example shows two *division by zero* alarms, D_{22} and D_{39}, of which D_{39} is a false positive, because the values of k at line 39 are non-zero. The assertions generated corresponding to these alarms for AFPE are respectively shown on lines 21 and 38. As the assertions are in different functions, they are not grouped together by the grouping techniques [8], [16]. For illustration purpose, we assume that the function *complexFunc* is complex, and verification of a code that contains this function, using a model checker, times out, i.e., given a sufficiently large time limit, the verification cannot conclude whether the assertion holds or fails.

We use $mcall(\phi, f)$ to denote a call to the model checker, which verifies the assertion generated corresponding to an alarm ϕ, and in the context of given function f. We call this function f *verification context*. Since the verification context in a model checking call is specified as a function, we use the terms *context* and *function* interchangeably. On similar lines, we use $scall(\phi, f)$ to denote a call to the slicer, which slices the code in function f with respect to the assertion generated corresponding to an alarm ϕ, assuming f as the entry-function. We use $S_{\phi,f}$ to denote the slice resulting from $scall(\phi, f)$, and $S_{\phi,App}$ to denote the slice generated when f is the entry-function of the application.

```
1  int t, x, y, z;
2  void main()
3    x=lib1(); y=lib2();
4    f1();
5  }
6
7  void f1(){
8    int i = 0;
9    while(i++ < x){
10     if(y == z){
11       f2(i);
12       f4();
13     }
14   }
15 }
16
17 void f4(){
18   t = complexFunc(0);
19   z = lib4();
20   if(z == 4){
21     //assert(x!=0);
22     display(1/x);        D22
23   }
24 }
```

```
21 void f2(int p){
22   int i, j;
23
24   i = p % 5;
25   j = lib4() % 5;
26   y = complexFunc(5);
27
28   f3(i,j);
29 }
30
31 void f3(int i, int j){
32   int arr[5]={4,2,3,9,0};
33   int k = 2;
34
35   if(i != j)
36     k = arr[i] - arr[j];
37
38   //assert(k!=0);
39   t = 100 / k;           D39
40 }
41
42 int complexFunc(int r){
43  ...//complex code
44 }
```

Fig. 1: Example to illustrate the need of combination of techniques for scalability of AFPE, and redundant slicing calls resulting in this combination.

B. The Need of Combination of Techniques

In AFPE, to use a model checker in a more scalable way, Post et al. [10] have proposed verification of assertions using approach called *incremental context expansion* (VerICE). Verification of each assertion (or a group of related/similar assertions [12], [16]) is started from the function that includes the assertion(s). Then the verification context 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 application's entry-function. This technique, VerICE, has been later adopted and found useful by several other works [8], [11], [12], [16]. During AFPE with VerICE, processing of D_{39} results in two model checking calls, first $mcall(D_{39}, f3)$ and then $mcall(D_{39}, f2)$, without eliminating the alarm as a false positive. The first call results in a counter-example (i.e., shows that the assertion fails), while the second call times out, because the function *complexFunc* is a part of $f2$.

During AFPE, to remove such complex code from code to be verified by the model checker, backward slicing [17] is used [9], [11], [12]. For each assertion or a group of related assertions, the complete application code is sliced with respect to the assertion(s), and the assertions in the sliced code are verified. However, usage of slicing at the application-level may not always help. For example, during AFPE with VerICE and application-level slicing, processing of D_{39}, still results in the same two model checking calls, generating the same results. It is because the function *complexFunc* is still a part of $f2$ in the application-level slice. AFPE with those two techniques also does not eliminate the false positive, D_{39}.

To address this problem, *verification context-level slicing* (that we call vcSlicing) is used [8], [11]. In this technique,

before each model checking call *mcall*(ϕ, f), the code is sliced using *scall*(ϕ, f) and the slice generated is verified by the model checker. For example, during AFPE with VerICE and vcSlicing, D_{39} gets eliminated as a false positive. In this case, the second model checking call, *mcall*(D_{39},*f2*), proves that the assertion holds, because the context-level slice generated, $S_{D_{39},f2}$, does not include the complex function.

To speed up vcSlicing, the application code is sliced at the application-level for each alarm/group of alarms, and then the assertions in each slice are verified using VerICE and vcSlicing [8], [11], [12]. Moreover, generating the application-level slice first also can be a requirement when other techniques are used, e.g., abstraction of unbounded loops based on the assertion(s) [18]. During AFPE with this combination of techniques, processing of D_{39} results in three slicing calls and two model checking calls: *scall*(D_{39}, *main*), *scall*(D_{39},*f3*), *mcall*(D_{39},*f3*), *scall*(D_{39},*f2*), and *mcall*(D_{39},*f2*). Processing of D_{22} results in three slicing calls and three model checking calls: *scall*(D_{22}, *main*), *scall*(D_{22},*f4*), *mcall*(D_{22},*f4*), *scall*(D_{22},*f1*), *mcall*(D_{22},*f1*), *mcall*(D_{22}, *main*). This indicates that the combination of techniques during AFPE increases the number of slicing and model checking calls. Henceforth, unless stated otherwise, we assume that AFPE is performed with the combination of those three techniques: application-level slicing, VerICE, and vcSlicing.

III. Computation of Redundant Slicing Calls

This section presents our technique to compute redundant vcSlicing calls resulting during AFPE.

A. Our Observation: Redundant vcSlicing Calls

Recall that, the goal of vcSlicing, i.e., any *scall*(ϕ, f), is to remove any (complex) code that belongs to f in $S_{\phi,App}$ and does not impact the result of *mcall*(ϕ, f). For example, the second vcSlicing call for D_{39} shown in Figure 1, *scall*(D_{39},*f2*), removes the function *complexFunc* from $f2$ present in $S_{D39,App}$. This code removal allows to eliminate the alarm as a false positive through *mcall*(D_{39},*f2*) (Section II-B). Note that, vcSlicing may not always result in such code removal, e.g., *scall*(D_{39},*f3*) does not remove any code from $f3$ present in $S_{D39,App}$. Moreover, the two vcSlicing calls for D_{22}, *scall*(D_{22},*f4*) and *scall*(D_{22},*f1*), also do not remove any code respectively from $f4$ and $f1$ present in $S_{D22,App}$. That is, among those total four vcSlicing calls, three are redundant, and they can be skipped to reduce the AFPE time.

B. Proposed Technique

1) Notations: Let *callers*(f) denotes the set of functions that call f directly or indirectly; and *callees*(f) denotes the set of functions that are directly or indirectly called by f, inclusive of f. A variable v at a program point p is said to be *data dependent* on a definition (assignment) d_v of v, if d_v is a reaching definition of v at p [19]. Data dependencies of a variable v are the definitions on which v is data dependent.

2) The Key Idea: We observe that, a vcSlicing call *scall*(ϕ, f) removes code belonging to f present in $S_{\phi,App}$ only if the following condition is satisfied: a definition in $g1 \in$ *callees*(f) is *a data dependency* of a variable used in $h \in$ *callers*(f), and the same definition is not a data dependency for any variable used in $g2 \in$ *callees*(f). That is, the call is useful (resp. redundant) if the condition is satisfied (resp. not satisfied). For example, *scall*(D_{39},*f2*) gets identified as useful because the condition is satisfied: the definition of y in $f2$ (at line 26) is a data dependency of y used in $f1 \in$ *callers*$(f2)$ (at line 10), and the same definition is not a data dependency for any variable used in any function in *callees*$(f2)$. We stress that, the condition *is not satisfied* for *scall*(D_{22},*f4*): the definition of z in $f4$ (at line 19) is a data dependency of z used in $f1 \in$ *callers*$(f2)$ (at line 10), but the same definition is also a data dependency of z used in $f4$ at line 20. Thus, *scall*(D_{22},*f4*) is redundant. For *scall*(D_{39},*f3*) and *scall*(D_{22},*f1*), the condition identifies them as redundant.

3) Computation Method: To efficiently compute whether the condition is satisfied for a given vcSlicing call *scall*(ϕ, f), we use the *program dependence graph* (PDG) [19] constructed while generating the application-level slices for the assertions. Starting from the assertion generated for ϕ, we traverse over the transitive data and control dependencies of variables in the assertion, and determine whether the condition is satisfied.

To avoid switching between the AFPE process and computation of redundant vcSlicing calls, instead of computing whether the corresponding vcSlicing call is redundant before each model checking call (on-demand computation), we first identify all possible vcSlicing calls and compute redundant calls from them. We use the computation results later when calls to the vcSlicing are actually to be made. We plan to design an aggressive variant of the technique, in which we will use heuristics to determine whether the code being removed by a vcSlicing call is potentially complex. The call will be identified as useful only if the code is determined as complex, otherwise redundant.

IV. AFPE on Partitioned-Code

This section presents our technique to improve efficiency of AFPE applied to alarms generated on partitioned-code.

A. Code Partitioning: An Example

Code partitioning is commonly used to scale static analysis tools to very large systems [13]–[15]. Each partition formed, is denoted by the single *sub-routine* (function) that is *entry* to the code that implements the functionality. Recall that two or more partitions can overlap (Section I). For example, Figure 2 shows two partitions *p1* and *p2*, and function *foo* is common to these partitions. For the same array access at line 17, an *array index out of bound* alarm gets generated at line 17, gets generated both the partitions, $p1$ and $p2$. These two alarms are respectively shown as A_{17}^{p1} and A_{17}^{p2}. Note that, although these two alarms are generated for the same *point of interest* (POI), they are different. We call such alarms *common-POI*

```
1  // Partition p1
2  void p1(){
3    foo(lib1());
4  }
5
6  // Partition p2
7  void p2(){
8    foo(lib2());
9  }
```

```
10 // Common function
11 void foo(int i){
12   int t, arr[4]={...};
13
14   assert(i>=0 && i<4
15 // common-POI alarms
16
17   arr[i] = 0;
18 }
```

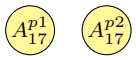

Fig. 2: Examples of partitioned-code and common-POI alarms.

TABLE I: Experimental results for improvement efficiency of AFPE on partitioned-code.

Application	Partitions (App. size–KLOC)	Property	Alarms	Groups of related assertions	False Positives Elimtd.	model checking calls	% Reduction in calls	AFPE Time (mins)	% Time Reduction.
smp_utils	29 (16)	AIOB	178	43	20	88	10.2	866.4	6.8
		OFUF	1534	277	198	210	11.4	1000.2	0.5
dict_gcide	8 (10.8)	AIOB	106	26	1	47	55.3	22.0	55.9
		DZ	128	100	8	94	58.5	23.2	40.6
		OFUF	652	331	43	139	33.8	40.3	19.7
uucp	5 (73.7)	AIOB	22	9	0	6	33.3	66.4	31.7
		OFUF	343	136	0	23	21.7	213.65	20.0
ffmpeg	93 (83.7)	AIOB	1775	507	340	589	6.8	904.2	11.0
		DZ	582	134	5	192	19.8	211.9	16.8
		OFUF	12375	1979	1314	1849	6.9	4217.4	7.3
Industry app.	3 (14.6)	AIOB	145	38	52	120	9.2	236.8	1.8
		DZ	9	6	0	18	0	69.1	0
		OFUF	300	89	75	208	15.9	452.8	2.4

alarms. The assertion generated corresponding to these alarms, for AFPE, is shown at line 14.

B. Our Observation: Redundancy in AFPE

We observe that, AFPE applied to common-POI alarms results in repeated slicing and model checking calls. For example, consider the two common-POI alarms in Figure 2. Processing A_{17}^{p1} results in two slicing and two model checking calls, $scall(A_{17}^{p1}, p1)$, $scall(A_{17}^{p1}, foo)$, $mcall(A_{17}^{p1}, foo)$, and $mcall(A_{17}^{p1}, p1)$. On similar lines, processing A_{17}^{p2} results in two slicing and two model checking calls, $scall(A_{17}^{p2}, p2)$, $scall(A_{17}^{p2}, foo)$, $mcall(A_{17}^{p2}, foo)$, and $mcall(A_{17}^{p2}, p2)$. Note that, for these two alarms, the vcSlicing and model checking calls made for the context *foo* are same. That is, the slices generated by $scall(A_{17}^{p1}, foo)$ and $scall(A_{17}^{p2}, foo)$ are same, and therefore, the results of $mcall(A_{17}^{p1}, foo)$ and $mcall(A_{17}^{p2}, foo)$ also are same. Since the function of a set of common-POI alarms is common to two or more partitions, processing the common-POI alarms during AFPE results in at least one repeated model checking call, and the corresponding vcSlicing call(s) are also repeated. Eliminating this redundancy helps to reduce the time taken by AFPE.

C. Reuse-based Technique for Efficiency Improvement

To eliminate redundancy in processing of common-POI alarms during AFPE, we implement a reuse-based technique similar to *memoization* [20] and *tabling* [21]. That is, before making a model checking call and its corresponding vcSlicing call, we check whether the same model checking 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 their corresponding vcSlicing calls, thus improving efficiency of AFPE. For example, applying this reuse-based approach to alarms shown in Figure 2 allows to reduce one model checking (resp. slicing) call out of the total four model checking (resp. slicing) calls.

V. Experimental Evaluation

The first technique to skip redundant slicing calls (Section III) is currently under evaluation. This section presents our initial evaluation of the reuse-based technique applicable for common-POI alarms (Section IV).

a) Implementation: To evaluate the improvement in AFPE efficiency due to our technique, we implemented AFPE with the following techniques: grouping of related assertions, abstraction of unbounded loops, slicing at application- and verification context-levels, and VerICE. We used CBMC [22] as the model checker: it is commonly used model checker in earlier AFPE-related works [8], [9], [11], [12]. For each model checking call, we set the time out threshold to 10 minutes.

b) Selection of Applications and Alarms: Table I presents five partitioned-code applications that we selected for the evaluation, along with the the number of partitions on them. The first four applications are open-source whereas the last one is an industry application from automotive domain. The last three applications are from benchmarks we used earlier [23], and first two were randomly chosen from applications available to us and having multiple partitions on them. We analyzed the partitions using our commercial static analysis tool for three properties: *array index out of bounds* (AIOB), *division by zero* (DZ), and *arithmetic overflow underflow* (OFUF). Column *Alarms* presents the number of alarms generated.

c) Evaluation Results: To evaluate the improvement in efficiency, we processed the alarms in two settings: with and without reusing AFPE results of common-POI alarms across partitions. Table I provides the results of AFPE: the number of *groups of related assertions*, *false positives eliminated*, the number of *model checking calls* made, *% reduction in model checking calls* due to our technique, time taken by AFPE (column *AFPE time*), and the *% time reduction* due to our technique. Note that, higher reduction in the number of model checking calls does not imply a similar reduction in the time. This occurs because the total AFPE time also includes the time required to process the alarms which are not common-POI alarms, generate *partition-level slices*, and over-approximate the loops. Our technique does not reduce this processing time.

The results indicate that, our proposed technique reduces the number of model checking calls by up to 58.5%, with median reduction of 19.8%. The reduction in calls reduced the AFPE time by up to 56%, with median reduction of 12.15%.

VI. Related Work

Existing techniques that improve AFPE efficiency are based on multiple approaches. The first prominent approach is to group related assertions and verify assertions in each group together [12], [16]. For example, Chimdyalwar and Darke [12] have used data and control dependencies to form groups of assertions that are related and present in same function. However, processing a group of related assertions during AFPE still can result in redundant slicing and model checking calls, e.g., when the code is partitioned (Section IV-B).

Another approach used to improve AFPE efficiency is to predict result of a given model checking call [11], [16], and use the predicted results to skip a subset of model checking calls. For example, Muske and Khedker [11] have used static analysis-based technique to predict model checking calls that are most likely to generate counter-examples. Since the calls generating counter-examples do not help to eliminate a false positive, they can be skipped, and the verification context can be expanded to the next level as per VerICE. Wang et al. [9] have used slicing as a mean to improve efficiency of AFPE.

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. This indicates that existing techniques and our both the techniques, being orthogonal to each other, can be used together.

VII. Conclusions & Future Work

Reducing the cost of redundant manual inspection of false positives, by eliminating them before the inspection, is important for wider adoption of static analysis tools. The model-checking based AFPE proposed for this goal, being computation-intensive, requires considerable time to postprocess the large number of alarms. Reducing this processing time is important, more particularly when the systems being analyzed are too large. Based on our observation that, even after applying the existing relevant techniques AFPE suffers from the problem of redundant calls to the slicer and model checker, we designed two automated techniques. We designed the techniques by taking into account (1) the particular type of common-POI alarms generated on partitioned-code, and (2) checking for a set of assertions and a context, whether the context-level slicing removes any irrelevant code from the context appearing in the application-level slice.

Our initial evaluation of the technique, applicable to common-POI alarms, has shown promising results: reduction in AFPE time by up to 56%, with median reduction of 12.15%. Detailed evaluation of both the techniques is currently ongoing. We believe that, on similar lines to the design of the proposed techniques, more techniques can be designed to reduce AFPE time. For example, we plan to design a technique to predict result of a model checking call based on the results of model checking calls made earlier for the same context but for some other alarms. Also, we plan to explore machine learning techniques for these predictions.

References

[1] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering*. IEEE, 2015, pp. 598–608.

[2] L. N. Q. Do, J. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, 2020.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *International Conference on Software Engineering*. IEEE, 2013, pp. 672–681.

[4] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *International Conference on Automated Software Engineering*. ACM, 2016, pp. 332–343.

[5] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 470–481.

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

[7] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *Conference on Programming Language Design and Implementation*. ACM, 2012, pp. 181–192.

[8] P. Darke, B. Chimdyalwar, A. 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*. IEEE, 2017, pp. 468–475.

[9] L. Wang, Q. Zhang, and P. Zhao, "Automated detection of code vulnerabilities based on program analysis and model checking," in *International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 165–173.

[10] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *International Conference on Automated Software Engineering*. IEEE, 2008, pp. 188–197.

[11] T. Muske and U. P. Khedker, "Efficient elimination of false positives using static analysis," in *International Symposium on Software Reliability Engineering*. IEEE, 2015, pp. 270–280.

[12] B. Chimdyalwar and P. Darke, "Statically relating program properties for efficient verification," in *International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, 2018, pp. 99–103.

[13] S. Khare, S. Saraswat, and S. Kumar, "Static program analysis of large embedded code base: An experience," in *India Software Engineering Conference*. ACM, 2011, pp. 99–102.

[14] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, 2008.

[15] T. Muske, "Improving review of clustered-code analysis warnings," in *International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 569–572.

[16] T. Muske, A. Datar, M. Khanzode, and K. Madhukar, "Efficient elimination of false positives using bounded model checking," in *International Conference on Advances in System Testing and Validation Lifecycle*. IARIA XPS Press, 2013, pp. 13–20.

[17] M. Weiser, "Program slicing," in *International Conference on Software Engineering*. IEEE, 1981, pp. 439–449.

[18] P. Darke, B. Chimdyalwar, R. Venkatesh, U. Shrotri, and R. Metta, "Over-approximating loops to prove properties using bounded model checking," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2015, pp. 1407–1412.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[20] U. A. Acar, G. E. Blelloch, and R. Harper, "Selective memoization," in *Symposium on Principles of Programming Languages*. ACM, 2003, pp. 14–25.

[21] T. Swift, "Tabling for non-monotonic programming," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 201–240, 1999.

[22] CBMC, http://www.cprover.org/cbmc/.

[23] T. Muske, R. Talluri, and A. Serebrenik, "Reducing static analysis alarms based on non-impacting control dependencies," in *Asian Symposium on Programming Languages and Systems*. Springer, 2019, pp. 115–135.