Efficient Elimination of False Positives using Static Analysis

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

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

***Abstract*—Bug detection using static analysis has been found useful in practice for ensuring software quality and reliability. However, it often requires sifting through a large number of warnings. This can be handled by generating an assertion corresponding to each warning and verifying the assertion using a model checker to classify the warning as an error or a false positive. Since model checking over larger code fragments is non-scalable and expensive, it is useful to model check a given assertion with a small calling context length. For this, the variables receiving values from outside the context are initialized with arbitrary values generated by non-deterministic choice functions. The calling context length is then gradually increased on a need basis to include callers higher up in the call chains. While this aids scalability by keeping the calling context as small as possible, it requires multiple calls to model checker for the same assertion, requiring a considerable amount of time.**

We present a static analysis to expedite false positive elimination. It is based on the following observation: When the variables involved in an assertion are allowed to take arbitrary values at the point of assertion, the assertion is most likely to be violated by some or the other combination of values. In such cases, usage of a model checker is redundant as it does not aid in resolution of the corresponding warning. Our data flow analysis identifies (an over-approximated set of) such variables using a novel lattice so that model checking of assertions involving such variables can be avoided. Our empirical evaluation demonstrates that, on an average, the proposed static analysis avoids 49.49% of the total model checking calls, and it reduces the false positives elimination time by 39.28%. However, this gain is achieved at the cost of missing 2.78% false positives which could have been eliminated otherwise.

I. Introduction

Static analysis [10], [13] and model checking [2], [5] have widely been used in achieving software quality and reliability by facilitating early detection of defects. These two techniques are complementary on the metric of scalability, precision, and efficiency [7], [16], [18]. Static analysis is able to verify very large and complex systems whereas model checking often faces scalability issue as the system size and complexity increases. Further, static analysis generates many false warnings, commonly referred to as false positives [7], whereas model checking is precise in property verification when it scales. Also, the software verification using static analysis is generally far more efficient as compared to the verification using model checking.

Given that verification problems are undecidable in general, manual review of warnings is inevitable to fulfill the practical needs of constructing trusted software. The cost of manual review can be reduced by increasing the precision of verification on the one hand and scaling it to larger programs on the other hand. Given the complementary nature of static analysis and model checking, some approaches [3], [8], [17], [19] try to combine the two to get the best of both the worlds: the scalability of static analysis and the precision of model checking. Among various combinations attempted so far, a cascaded approach [6], [14] uses static analysis first and then the generated warnings are processed by a model checker to eliminate the false positives. This processing involves:

- Generating an assertion corresponding to each warning, where the assertion relates to the property (correctness or safety) being checked at the warning point. That is, a warning is a false positive (the warning point is *safe*) when its corresponding assertion holds.
- Verifying the generated assertion using a model checker to reduce the list of warnings needing manual review.

This approach of combining static analysis with modeling checking has been found to be very effective in eliminating a significant number of false positives without being affected by the non-scalability of the model checking [6], [14], [12]. It uses the following insight for scaling model checking: Since model checking over larger code fragments is impractical, it is useful to model check a given assertion with a small calling context in which the variables receiving values from outside the context are initialized with arbitrary values generated by non-deterministic choice functions [6] (these functions are described in detail in Section III-A). The calling context is then incrementally expanded on a need basis to include callers higher up in the call chains (as described in Algorithm 1 in Appendix). While this aids scalability by keeping the calling context as small as possible, it requires multiple calls to model checker for the same assertion which ironically increases the time required [12]. The following factors contribute to the inefficiency of the overall approach:

i) Static analysis issues a large number of warnings due to its imprecision.
ii) Model checking is expensive due to the state space explosion.
iii) Avoiding the state space explosion (and in turn non-scalability) in model checking by incremental expansion of calling contexts generally leads to multiple calls to model checker for the same assertion.

In this paper, we address the third source of inefficiency

```
1.    const int arr[] = {0, 2, 5, 9, 14};
2.    int ch, var, factor;
3.
4.    void f1(){
5.       unsigned int i, j;
6.
7.       i = lib1();
8.       j = lib2();
9.       var = lib3();
10.
11.      if(i < j && j < 5){
12.         factor = arr[j] - arr[i];
13.         f2();
14.      }
15.   }
```

```
21.   int f2(){
22.    if(var == factor)
23.       f3(var);
24.       ...
25.   }

31.   int f3(int p){
32.    int a, b, denom = 1;
33.    if(ch < 5)
34.       denom = p;
35.    else
36.       denom = 10;
37.
38.    assert(denom!=0);
39.    a = 100 / denom;//warning
40.
41.    if(b < 10)
42.       ch = p;
43.}
```

lib1, *lib2*, and *lib3* are library functions whose code is not available for static analysis. The return-type of *lib1*, and *lib2* is *unsigned int*, and for *lib3*, it is *signed int*. The assertion at line 38 has been inserted to resolve the *divide by zero* warning reported at line 39.

Fig. 1: Motivating example for false positive elimination.

by trying to minimize the number of model checker calls for an assertion by eliminating the redundant calls. Incremental expansion of calling contexts is required when the model checking in smaller context finds a counterexample. We observe that, in most cases the counterexample is generated not because of the values assigned to variables by the statements in the program but by the non-deterministic choice functions introduced outside of the context. That is, when the variables involved in an assertion are allowed to take any arbitrary value at the point of assertion, the assertion is most likely to be violated by some or the other combination of values. In such cases, using a model checker does not aid in resolution of the corresponding warning and hence is redundant. We are more likely to resolve a warning by model checking the assertion in an expanded context in which variables are not allowed to take all possible values at the point of assertion.

We propose concept of Complete-range Non-deterministic Values (*cnv*) variables to denote the variables taking any arbitrary value (all possible non-deterministic values), and use these to identify redundant model checking calls. In general, computing *cnv* variables is undecidable because of the static approximation of execution paths by control flow paths. Therefore, we can only compute an approximation of *cnv* variables. There are two kinds of approximations in play: (a) over-approximation of execution paths by control flow paths, and (b) over-approximation or under-approximation of the set of *cnv* variables at a program point as a *may* or a *must* property. The first kind of approximation is inevitable. We have defined a data flow analysis with a novel lattice structures for the second kind of approximations. Due to these approximations,

- we may end up eliminating more model checking calls than we ideally should.
- we may end up missing some redundant model checking calls that we ideally should not.

We have studied these effects empirically and found them to be relatively insignificant compared to the benefits. In particular, our empirical evaluation using two embedded system applications (of sizes 40 KLOC and 50KLOC) indicated that, on an average, the proposed static analysis identifies model checking calls violating the assertion with a precision of 97.3%. This identification reduces the total model checking calls by 49.49% and the false positives elimination time by 39.28%. However, it is achieved at the cost of missing 2.78% of false positives which could have been eliminated otherwise. This elimination loss is due to the imprecision in computation of *cnv* variables. The empirical evaluation also demonstrated trade-off between efficiency and precision in false positives elimination when the *cnv* variables are computed at different levels of over-approximation.

This paper makes the following research contributions.

- It introduces the notion of *cnv* variables.
- It presents a light weight data flow analysis to compute the *cnv* variables, and uses these variables to identify redundant modeling checking calls during elimination of false positives.
- It exploits the trade-off between the precision and effciency of false positives elimination and demonstrates the possibility of improving efficiency considerably with a negligible loss in precision.

Outline: Section II provides a motivating example. In Section III, we present a static analysis to identify redundant model checking calls by computing the *cnv* variables. Details about the implementation, experimental set up and results are provided in Section IV. Section V presents related work, and finally we conclude in Section VI with future work.

II. A Motivating Example

Consider the example in Figure 1 whose verification using static analysis reports a *divide by zero* warning at line 39. This warning is a false positive: The only way the variable *denom* can be zero is if the variable *p* is zero (line 34). This requires the actual argument *var* of the call to *f3* to be zero (line 23) which depends on the values of the variable *factor*

(line 22). Given the initialization of array *arr* (line 1), for any combination of index variables satisfying $i < j$, the RHS of the assignment at line 12 can never be zero, ruling out any possibility of *factor*, and hence *denom*, being zero.

In order to handle this warning, an assertion has been added (line 38). Let A_n denote an assertion at line *n*, and $V(A_n, f)$ denote its verification in the calling context beginning with procedure f. Then model checking of A_{38} using the approach of context expansion proceeds as follows:

i) The first call to model checker is $V(A_{38}, f3)$ with non-deterministic values assigned to the variables receiving values from outside the context of $f3$. These values are assigned at the start of $f3$ (shown in Figure 2 and also described in Section III-A). The model checker trivially finds a counterexample by choosing the value of *p* as 0.
ii) The next call $V(A_{38}, f2)$ expands the calling context with non-deterministic values assigned to the variables receiving values from outside the context of $f2$ (shown in Figure 2). The model checker once again reports a counterexample by choosing *var* and *factor* to be 0.
iii) The third call $V(A_{38}, f1)$ assigns non-deterministic values to the variables receiving values from outside the context of $f1$ (shown in Figure 2). Now regardless of the values of *i* and *j*, the model checker fails to find value 0 for *factor* (and in turn to *var* and *p*), and hence it declares that the assertion A_{38} always holds. Thus, the verification call $V(A_{38}, f1)$ eliminates the warning at line 39.

It is easy to see that the model checker is invoked multiple times for the same assertion. Further, the first and second calls do not contribute in eliminating the false positive. These calls generate counterexamples because the values assigned by non-deterministic choice functions reach line 38 (the assertion point) unconstrained. That is, variable *denom* is a *cnv* variable at line 38 for the model checker calls $V(A_{38}, f3)$ and $V(A_{38}, f2)$. Hence these calls are provably redundant.

Note that the property that a variable may take any arbitrary value from its complete range of values is orthogonal to whether it is initialized or not. For example, in Figure 1:

- Variable *i* has been initialized on line 7 but it is a *cnv* variable at line 10 because the library function *lib*1 may return any value. On the other hand the same variable is not a *cnv* variable at line 12 because it cannot take any value greater than 3.
- Variable *b* is uninitialized in procedure $f3$. It is a *cnv* variable on lines 33 to 39 but not on line 42 because it cannot take any value greater than 9 when the execution reaches line 42.

Thus the information computed by *cnv* variables analysis is incomparable with the information computed by a possibly uninitialized variables analysis and the two analyses are different.

III. A Static Analysis for RVCs Identification

This section describes our approach of identifying redundant verification calls through computation of *cnv* variables in the program. We begin by explaining some terms and concepts by referring to the code sample in Figure 1.

A. Basic Terms and Concepts

1) Assertion Variables: A given assertion A_n is a boolean expression describing a constraint on the values of the variables occurring in the expression. This constraint must be satisfied every time statement *n* is executed. The variables occurring in an assertion are called *assertion variables*. For example, *denom* is the only assertion variable used in A_{38}.

2) Input Variables: The computations in a procedure could depend on global variables and formal parameters which take values from the calling contexts. Such variables are termed as *input variables* of the procedure. A variable *v* is identified as an input variable of procedure f if there exists a path in f, on which *v* is read before it is written to. Since we consider interprocedural paths starting in f, they include the paths in callees of f transitively.

Note that the input variables

- are procedure specific (e.g. *factor* is an input variable for $f2$ but not for $f1$),
- may be indirectly accessed in a (transitive) callee (e.g. *ch* is an input variable for both $f1$ and $f2$ though it is read only in $f3$), and
- must be mutable variables (e.g. *arr* is not an input variable for $f1$ because it is a *constant* array.)

3) Non-deterministic Choice Functions: As mentioned in Section II, verification of assertion A_n in the context of procedure f is denoted by $V(A_n, f)$ where A_n appears in a statement either in f or one of its transitive callees. For scalability of verification, f is chosen as close to the assertion in the call hierarchy as possible and the callers of f are ignored. However, this comes at the cost of precision—since the code assigning values to input variables of f is ignored, A_n should be verified for arbitrary values of the input variables. If the assertion is verified successfully, it holds for any arbitrary value and hence by implication, for the original values in the program.

For this purpose, arbitrary values are assigned to the input variables by using non-deterministic choice functions. Figure 2 lists the non-deterministic choice functions assigning values to the input variables of the procedures in Figure 1. The range of the non-deterministic values generated by these functions is determined by their return-types (not shown).

During a verification call $V(A_n, f)$, a model checker generates all possible combinations of values of the input variables of f, and evaluates the assertion A_n for every combination. If the assertion is not violated by any of the combinations, the model checker reports successful verification of the assertion. Otherwise, the model checker provides a counterexample in the form of program trace that violates the assertion. Henceforth in the paper, the terms *counterexample* and assertion violation are used interchangeably.

4) Complete-range Non-deterministic Value (cnv) Variables: The values assigned to input variables of a procedure by the non-deterministic choice functions, at the start of the procedure, further may get assigned to other variables in the procedure as a consequence of data-dependence. These values also may get constrained through control or data-dependence. Let Rn_p be the range of the non-deterministic values that a variable *v* may take at some program point *p*, and let *Ra* be

```
// procedure f1
ch = nondet_char();
```

```
// procedure f2
var = nondet_int();
ch = nondet_char();
factor = nondet_int();
```

```
// procedure f3
p = nondet_int();
ch = nondet_char();
```

Fig. 2: Examples of functions assigning non-deterministic values

the range of values that v is allowed to take at any program point (that is, Ra is the range of data-type of v). A variable v is called a *cnv* variable at a program point p only if $Ra = Rn_p$. Thus, we categorize the variables taking the complete range of the non-deterministic values as *cnv* variables, and the other set of variables (taking partial set of such values or fixed values assigned by the program statements) as non-*cnv* variables.

The factors that influence computation of *cnv* variables are described below.

(a) *Context sensitivity.* The assignments involving non-deterministic choice functions are introduced at the start of the procedure that begins the calling context under consideration. Thus, a variable may be a *cnv* variable for a specific calling context but not for some other calling context. For example, *factor* is a *cnv* variable in the context of $f2$ at all program points belonging to procedure $f3$, since it directly receives the complete range of the correspondingly assigned non-deterministic values. However, when we extend the context to begin at $f1$, *factor* ceases to be a *cnv* variable at the same points (in procedure $f3$), because the values assigned at line 12 are computed using constant array elements.

(b) *Flow sensitivity.* The range of the values of a variable is program point-specific and it may be different at different program points. For example, *denom* is a *cnv* variable at the exit point of the statement at line 34, because it is assigned with the value of a *cnv* variable p. However, it is not a *cnv* variable at the exit point of the statement at line 36, since it is assigned with 10. Flow sensitivity influences the *cnv* status by the following two dependencies:

(i) *Data dependence*. A *cnv* (resp. non-*cnv*) variable at a program point may influence the values of a non-*cnv* (resp. *cnv*) variable and convert its status. For instance, with $f3$ as the calling context, the assignment statement at line 34 changes status of *denom* from non-*cnv* to *cnv*.

(ii) *Control dependence*. The conditions involving a *cnv* variable may restrict the values of the variable along their *true/false* branches. For example *ch* is a *cnv* variable before line 33 but becomes a non-*cnv* variable on lines 34 and 36, because the condition in line 33 constrains the range of values of *ch*. Further, the values of *ch* at these points (entry points of statements at lines 34 and 36) are complementary. We use this fact later in our analysis to retrieve the *cnv* status of *ch* at the later points (e.g. points at lines 38 and 39) which are not under the influence of the same condition. That is, *ch* becomes a *cnv* variable again at these points (line 38 and 39).
Note that every condition may not restrict the value. For example, with $f2$ as the calling context, the condition at line 22 does not restrict the values of *var* or *factor*.

We use $cIn(n, f)$ (resp. $cOut(n, f)$) to denote the *cnv* variables at the entry (resp. exit) point of statement n and with procedure f as the calling context.

(c) *May/must reachability.* The *cnv* status of a variable v at a program point depends on the paths along which v is *cnv*. When v is *cnv* along every path reaching the program point, v is said to be a *must-cnv* at that point. If it is *cnv* along some but not necessarily all paths reaching the program point, it is a *may-cnv*. For example, when $cIn(38, f3)$ are computed as *must-cnv* variables, $ch, p \in cIn(38, f3)$ and $denom \notin cIn(38, f3)$. Further, when $cIn(38, f3)$ corresponds to *may-cnv* variables, $denom \in cIn(38, f3)$.

5) Complete-Range Non-deterministic Value Expressions: We extend the concept of non-deterministic value variables to expressions. An expression e occurring in statement n is said to be a *cnv* expression in the calling context of procedure f if its evaluation using the *cnv* variables at the same point (entry of n) and in the same context (f), results in complete range of values of e. For example, assuming variables x, y, and z to be integers,

- expressions $(x+10)$, $(x++)$, and $(x+y)$ in statement n are *cnv* expressions when $cIn(n, f) = \{x, y, z\}$.
- expressions $(x+10)$, and $(x++)$ in statement n are not *cnv* expressions when $cIn(n, f) = \{y, z\}$.
- expressions $(x/100)$, $(x\%2)$, and (100) in statement n are not *cnv* expressions when $cIn(n, f) = \{x\}$.
- the arithmetic expression $(arr[j] - arr[i])$ in statement at line 12 in Figure 1 is not a *cnv* expression with respect to any *cnv* variables.

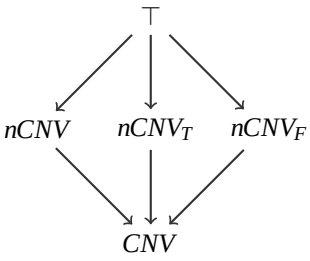
B. Computation of *cnv* Variables

We present a data flow analysis for identification of *cnv* variables in an intraprocedural setting which can be easily lifted to interprocedural setting.

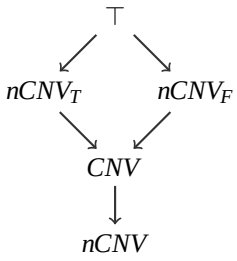
Let N be the set of nodes in the control flow graph of the program being analyzed, and V be the set of program variables. We define S = $\{CNV, nCNV, nCNV_T, nCNV_F\}$, as shades of the *cnv* status of a variable $v \in$ V at a node $n \in$ N, where

- CNV: v is a *cnv* variable.
- $nCNV$: v is not a *cnv* variable due to data dependence on a non-*cnv* variable or expression.
- $nCNV_T$ (resp. $nCNV_F$): v is not a *cnv* variable due to control dependence (when its values are constrained along the paths reachable from *true* (resp. *false*) branch of a condition).

We use following notational conventions:



(a) *may-cnv* variables



(b) *must-cnv* variables

Fig. 3: Lattices for computing *cnv* variables using data flow analysis

- Mapping stat = V $\mapsto$ S relates a variable $v \in$ V to its *cnv* status $s \in$ S.
- α ranges over the set $A = 2^{\text{stat}}$ and thus it represents a mapping from variables (V) to *cnv* status (S). $\alpha(v)$ returns the status of a variable v in the mapping α, and $\alpha[v \mapsto s]$ updates the status of v to s in the mapping α.
- Predicate isCNV ($expr$, α) asserts that expression $expr$ is a *cnv* expression with given a mapping set α.
- For $m, n \in$ N, $e = m \rightarrow n$ denotes an edge from node m to n. The label of this edge is denoted by $label(e)$ or $label(m \rightarrow n)$. We assume that
  - $label(e)$ = *, when e is an unconditional edge.
  - $label(e)$ = *true*, when e is the *true* branch of m.
  - $label(e)$ = *false*, when e is the *false* branch of m.
  - the conditional edge from *switch* to its first *case* is labeled as *true*, while the edges from the *switch* to its other *cases* are labeled as *false*.
  - Function $pred(n)$ returns the predecessor nodes of a given node n in the control flow graph.

1) Lattices.: Our analysis computes subsets of stat flow-sensitively at each node $n \in$ N, and the lattice of these values is $\langle A = 2^{\text{stat}}, \sqcap_A \rangle$. As stat = V $\mapsto$ S is defined in terms of lattice $(S, \sqcap_S)$, the meet operation $\sqcap_A$ is defined in terms $\sqcap_S$ as shown below.

$$\forall x, y \in A: \; x \sqcap_A y = \left\{ \left(v, \; \left(s \sqcap_S s'\right)\right) \;\middle|\; (v,s) \in x, \left(v,s'\right) \in y \right\} \quad (1)$$

Figure 3a (resp. 3b) presents the lattice $(S, \sqcap_S)$ to compute the *may-cnv* (resp. *must-cnv*) variables. The $\top$ element in the lattices is a fictitious value used as an initialization. The meet of $nCNV_T$ and $nCNV_F$ results in CNV. That is, status of v is restored after the effect of a controlling condition which has marked v to be $nCNV_T$ (resp. $nCNV_F$) along its *true* (resp. *false*) branch. The $\perp$ element in Figure 3a (computing *may-cnv* variables) is CNV indicating that v is a *cnv* variable along some path. The $\perp$ element in Figure 3b (computing *must-cnv* variables) is $nCNV$ indicating that v is not a *cnv* variable if it is not a *cnv* along any of the paths.

2) Data Flow Equations.: Figure 4 shows the data flow equations. Note that in Equation 7, the status of a variable v is changed to $nCNV_T$ (resp. $nCNV_F$) as an effect of condition $v \oplus expr$, if and only if (a) v has its status as CNV before to the condition, (b) $expr$ is not a *cnv* expression, *and* (c) the edge is labeled as *true* (resp. *false*).

```
1.  void foo(){
2.    b = 0;
3.    if(v == 1) {
4.      a = 10;
5.      b = 10;
6.    }
7.
11.   if(b == 10)
12.     assert(v != 0);
13.
14.   if(x < 100)
15.     assert(x < 50);
16.
17.   if(v == 1)
18.     assert(a != 0);
19.}
```

Fig. 5: Impact of *may*/*must-cnv* variables

C. Identification of Redundant Verification Calls

We identify an assertion verification call $V(A_n, f)$ as redundant when all assertion variables of A_n belong to *cIn*(n, f). Using this criterion, the calls $V(A_{38}, f3)$ and $V(A_{38}, f2)$ described in the motivating example (Section II) are identified as redundant verification calls (RVCs). This is because, *denom* $\in$ *cIn*$(38, f3)$ and *denom* $\in$ *cIn*$(38, f2)$ when computed as *may-cnv* variables. Further, the third call $V(A_{38}, f1)$ is not identified as an RVC since *denom* $\notin$ *cIn*$(38, f1)$. We prefer requiring all assertion variables to be *cnv* variables even though any one of them satisfying this constraint may be sufficient, because constraining over all assertion variables is a tighter constraint as compared to requiring any variable to be a *cnv* variable.

It is important to note that:

a) We can not guarantee that an RVC identified by using *cnv* variables will always result in a counterexample. For example, for the program in Figure 5, the $V(A_{12}, foo)$ is identified as an RVC since the assertion variable v is a *must-cnv* variable at line 12. This is a false identification because A_{12} always holds. Although the *cnv* variables may not identify all RVCs *correctly*, we expect a significant benefit in practice because such cases are rare in practice.
b) Further, a model checking call that is not identified as an RVC by the *cnv* variables, also can result in a counterexample. For example, the call $V(A_{15}, foo)$ in Figure 5 is not identified as redundant because x is not a *cnv* variable at line 15. However, this call always results in a conterexample.

This indicates the proposed RVCs identification approach is *imprecise*. Thus, to measure its effectiveness, we define two metrics for the RVCs identified by *cnv* variables.

Let $m,n \in \mathsf{N}$ and $u,v \in \mathsf{V}$

$$In_n = \begin{cases} \{(v,CNV) \mid v \in \mathsf{V}\} & n = StartNode \\ \bigsqcap_{m \in pred(n)}^{A} Edge_{m \to n}(Out_m) & \text{otherwise} \end{cases} \quad (2)$$

$$Out_n = update(In_n, n) \quad (3)$$

$$update(X,n) = \begin{cases} X[v \mapsto nCNV] & n : v = constant \\ X[v \mapsto X(u)] & n : v = u \\ Assign(X,v,expr) & n : v = expr \end{cases} \quad (4)$$

$$Assign(X,v,expr) = \begin{cases} X[v \mapsto nCNV] & isCNV(expr,X) = false \\ X[v \mapsto CNV] & isCNV(expr,X) = true \end{cases} \quad (5)$$

$$Edge_{m \to n}(X) = \begin{cases} X & \text{edge } m \to n \text{ is unconditional} \\ Cond(X,v,label(m \to n),expr) & \text{edge } m \to n \text{ is conditional, where } m : v \oplus expr \end{cases} \quad (6)$$

$$Cond(X,v,lbl,e) = \begin{cases} X[v \mapsto nCNV_T] & X(v) = CNV \text{ and } lbl = true \text{ and } isCNV(e,X) = false \\ X[v \mapsto nCNV_F] & X(v) = CNV \text{ and } lbl = false \text{ and } isCNV(e,X) = false \\ X & \text{otherwise} \end{cases} \quad (7)$$

Fig. 4: Data flow equations for computing *cnv* variables.

$$\text{Precision} = \frac{\text{number of } \textit{correctly} \text{ identified RVCs}}{\text{total number of identified RVCs}}$$

$$\text{Recall} = \frac{\text{number of } \textit{correctly} \text{ identified RVCs}}{\text{number of actual calls violating the assertions}}$$

1) Impact of may/must *approximation and over-approximation of execution paths:* The *may-cnv* variables identify a larger number of RVCs (higher recall) as compared to the RVCs identified by referring to the *must-cnv* variables, but with lesser accuracy (precision). To understand this difference, consider the following examples:

- Identifying more RVCs: As observed in the motivating example (in Section II), verification calls $V(A_{38}, f3)$ and $V(A_{38}, f2)$ are identified as RVCs by the *may-cnv* variables. However, they are not identified as redundant if we use the *must-cnv* variables because *denom* is not a *must-cnv* variable in any of the calling contexts.
- Lesser accuracy: For the code in Figure 5, the call $V(A_{18}, foo)$ is identified as an RVC by the *may-cnv* variables. This is a false identification because the call never results in a counterexample. This call is not identified as redundant by the *must-cnv* variables.

The influence of *may-* and *must-cnv* variables on RVCs is different because of the nature of approximations used in the two computations. A *may* analysis over-approximates the set of *cnv* variables whereas a *must* analysis under-approximates it. With over-approximation, we may declare more RVCs although with lesser accuracy because of the possibility of spurious inclusion of variables in the set. With under-approximation, we may identify fewer RVCs but with increased accuracy because every variable included in the set is more likely to be a *cnv* variable although we may have missed some genuine *cnv* variables.

To see the impact of over-approximation of execution paths, consider the example in Figure 5 for which *must-cnv* analysis discovers variable *v* to be a *cnv* variable on line 12, and $V(A_{12}, foo)$ as an RVC whereas *v* is guaranteed to be 1 on line 12. Since the condition on line 11 does not involve *v* (or *v* is not modified earlier), our analysis assumes that *v* is unconstrained. Effectively, it considers an execution path in which line 12 is executed but line 5 is not executed. Clearly no such execution path is possible because every path that passes through line 12 must necessarily pass though line 5 also.

2) Influence of cnv Analysis on False Positives Elimination: Since the set of *cnv* variables computed using data flow analysis is an over-approximation of *truly cnv* variables, some non-*cnv* variables (like *a* and *v* in Figure 5 at the assertion points A_9 and A_{12} respectively) may be reported as *cnv* variables. Due to such false reporting, we end up eliminating more model checking calls than we ideally should have. This has the effect of missing out on resolution of some warnings that could have been eliminated as false positives had we been able to compute set of *cnv* variables precisely. The consequences in false positives elimination (FPE) are illustrated in Figure 6. Hence, we define below two metrics for measuring the effect of skipped RVCs in efficient false positives elimination (*eFPE*) by comparing it with the original approach of false positives elimination (FPE_{orig}).

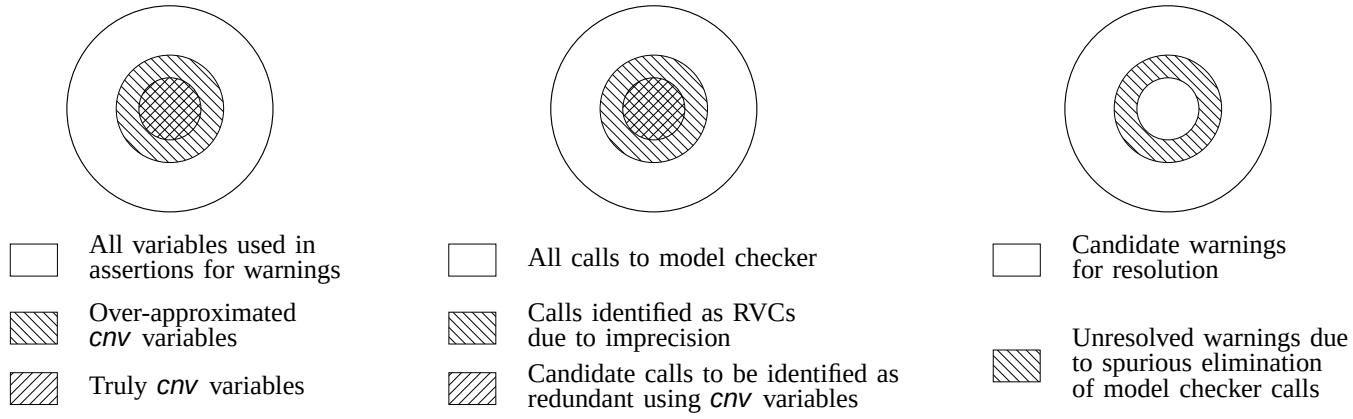


Fig. 6: Influence of *cnv* analysis on FPE. If *cnv* variables could be computed precisely, the ring between the two inner circles (shaded with ▨) in all the three pictures is most likely to vanish.

$$\text{Time saving} = 1 - \frac{\text{Time taken in } eFPE}{\text{Time taken by } FPE_{orig}}$$

$$\text{Elimination loss} = 1 - \frac{\text{False positives eliminated in } eFPE}{\text{False positives eliminated by } FPE_{orig}}$$

It is intuitive that the *may-cnv* variables will contribute more towards these metrics as compared to the *must-cnv* variables.

IV. Empirical Evaluation

This section describes our empirical evaluation of the proposed static analysis.

A. Implementation

We used TCS Embedded Code Analyzer (TCS ECA) [1] (a static analysis tool) to verify C code, and CBMC [4] to eliminate false positives from the generated warnings. CBMC is a Bounded Model Checker for C and C++ programs. We chose these tools for our experimentation because we have prior experience in their usage and they are integrated in the existing tool set [6], [12]. The tool set also includes a slicer supported by the analysis framework of TCS ECA, allowing slicing [11] the code with respect to an input assertion before it is verified using CBMC. We implemented a flow- and context-sensitive data flow analysis in the TCS ECA framework for computing *cnv* variables. These variables are computed in interprocedural setting by using procedure summaries.

B. Experimental Set-up

Applications: Two embedded system applications coded in C were selected: (i) a smart card management system (SCMS) of size 50 KLOC, and (ii) an automobile battery control module (BCM) of size 40 KLOC.

Verification Properties: The selected applications were verified for two properties — Divide by Zero (DZ), and Array Index Out of Bound (AIOB) as these are the most commonly checked properties.

TABLE I: Experimental results summary

Verification setting (warnings processed)	e-FPE settings	% model checking calls skipped	% counter-examples reduction	% time saving	% elimination loss
SCMS-DZ-1 (41)	$eFPE_{must}$	34.15	66.67	41.55	0.00
	$eFPE_{may}$	36.59	71.43	47.60	0.00
SCMS-DZ-5 (41)	$eFPE_{must}$	35.06	48.78	44.67	0.00
	$eFPE_{may}$	53.25	75.61	69.05	0.00
SCMS-AIOB-1 (137)	$eFPE_{must}$	45.99	56.76	38.35	0.00
	$eFPE_{may}$	83.21	100.00	71.81	0.00
SCMS-AIOB-5 (137)	$eFPE_{must}$	35.78	43.46	34.98	0.00
	$eFPE_{may}$	89.34	100.00	88.79	0.00
BCM-DZ-1 (39)	$eFPE_{must}$	5.13	7.14	5.37	4.55
	$eFPE_{may}$	5.13	7.14	7.84	4.55
BCM-DZ-5 (39)	$eFPE_{must}$	8.77	26.32	11.14	2.94
	$eFPE_{may}$	8.77	26.32	11.11	2.94
BCM-AIOB-1 (258)	$eFPE_{must}$	51.55	78.18	43.53	0.00
	$eFPE_{may}$	51.94	78.79	44.08	0.00
BCM-AIOB-5 (258)	$eFPE_{must}$	41.87	78.05	28.06	3.33
	$eFPE_{may}$	43.21	78.54	28.67	6.11
Over all runs of *eFPE* settings		49.49	72.87	39.28	2.78

False Positives Elimination (FPE) Settings: The FPE is carried out in three settings as described below.

a) FPE_{orig}: This is the original setting in which no model checking call is skipped as an RVC.
b) $eFPE_{must}$: This is the setting for efficient elimination of false positives in which RVCs are identified and skipped when all variables in an assertion are *must-cnv* variables.
c) $eFPE_{may}$: In this setting, RVCs are identified and skipped when all variables in an assertion are *may-cnv* variables.

Each of the above three FPE settings is further refined at two different levels by varying the maximum allowed calling context length (maxCCL): maxCCL = 1, and maxCCL = 5. This is because, in our experience, context length of 5 is generally sufficient when a warning can be resolved as a false positive and CBMC does not usually scale up beyond these

contexts. Further, in each setting, CBMC was made to time out after 120 seconds of verification time.

Assumptions: We have made (and ensured), the following assumptions for each of the above three FPE settings.

- We have restricted RVC identification to assertions that are reachable, since *cnv* variables are computed without performing reachability analysis. Thus, our analysis may report RVCs for model checking calls that are guaranteed to eliminate false positives corresponding to the unreachable assertions. This affects the precision of RVCs identification and loss in false positives eliminated. Hence in our experiments, to avoid this effect, we have manually identified reachable assertions and have restricted our measurements to them.
- The code to be verified is *sliced* separately with respect to each of the input assertions, so that every code slice corresponds to a single assertion.

Hardware Configuration: Our experiments are performed on a machine with Intel Core i7-4600U CPU @ 2.10 GHz 2.70 GHz, 8 GB RAM configuration, and having Windows 7 Enterprise SP1 as the operating system.

TABLE II: Details on identification of RVCs

Verification setting (warnings processed)	Counter example calls in FPE_{orig}	eFPE settings	Identified RVCs	Correctly identified RVCs
SCMS-DZ-1 (41)	21	$eFPE_{must}$	14	14
		$*eFPE_{must}$	15	15
		$eFPE_{may}$	15	15
		$*eFPE_{may}$	18	18
SCMS-AIOB-1 (137)	111	$eFPE_{must}$	63	63
		$*eFPE_{must}$	67	66
		$eFPE_{may}$	114	111
		$*eFPE_{may}$	115	111
BCM-DZ-1 (39)	14	$eFPE_{must}$	2	1
		$*eFPE_{must}$	2	1
		$eFPE_{may}$	2	1
		$*eFPE_{may}$	2	1
BCM-AIOB-1 (258)	165	$eFPE_{must}$	133	129
		$*eFPE_{must}$	140	132
		$eFPE_{may}$	134	130
		$*eFPE_{may}$	139	132

C. Experimental Results

The experimental results are presented in below described tables and figures, where a code verification setting is denoted by *application-property*-maxCCL (like SCMS-AIOB-1, and BCM-DZ-5).

a) Table I presents the number of warnings processed, and percentage of — (i) model checking calls avoided, (ii) reduced counterexample calls, (iii) time saved, and (iv) loss in eliminated false positives — in each of the *eFPE* settings. Further, it also presents these percentages computed over all runs of the *eFPE* settings (percentage of the total of the numbers in next described figures).

b) Figure 7 reports the time taken (in minutes with the fraction part ignored) by each of the FPE settings. The presented time is average time, computed from three different runs of the same FPE setting to minimize the effect of performance variations of the used machine. In these results, the time spent in slicing the code with respect to the input assertions is not included.

c) Figure 8 compares the false positives eliminated in each of the settings.

d) Figure 9 presents the total model checking calls made in each of the three FPE settings.

e) Figure 10 shows the number of calls that have resulted in counterexamples (assertion violations). In these results, the counterexamples reported due to insufficient loop unwinding [12] are not counted.

f) Table II presents details about the RVCs identified for the code verification settings with maxCCL=1. These details are also provided for two variants of the *eFPE* settings - $*eFPE_{must}$ and $*eFPE_{may}$- in which the RVCs are identified when any of the assertion variable is a *cnv* variable. Results of these settings are discussed in Section IV-D3.

D. Results Discussion (Observations)

Our measurements are summarized in tables I and II. We observe that:

a) The *eFPE* settings ($eFPE_{must}$ and $eFPE_{may}$), on an average, reduced the total model checking calls by 49.49%, and this in turn, reduced the false positives elimination time by 39.28%.

b) There is a large number (58%) of the model checking calls in FPE_{orig} that result in the counterexamples. However, these calls are reduced to 31% of the total model checking calls in the *eFPE* settings (refer to figures 9 and 10). Also, on an average, the model checking calls that resulted in counterexamples are reduced by 72.87%.

c) Overall, there are 21 false positives (out of 754) that were eliminated in FPE_{orig} but not in the *eFPE* settings. That is, the above speed up (39.28%) is achieved at the cost of elimination loss of 2.78%. Our manual analysis of these 21 cases indicated that, their corresponding model checking calls are identified as redundant (when they were not) due to the imprecisely computed *cnv* variables. Had we been able to compute the *cnv* variables more precisely, the loss would have been close to zero.

d) Referring to Table II, on an average, the *must-cnv* and *may-cnv* variables identify 76% of the total model checking calls resulting in counterexamples (recall=76%) with a precision of 97.3%.

e) The time taken to identify the RVCs was negligible as compared to the time saving achieved.

1) Trade-off: Table I shows that it is always possible to reduce the total time spent in model checking the assertions although the quantum of reduction may vary from minutes to hours. However, this performance improvement comes at the cost of missing elimination of 21 warnings which need to be manually inspected. The time spent in this depends on various factors like type of warnings, code complexity, reviewing skills of the user, and tool support used. This demonstrates an interesting tradeoff: on the one hand, we could let the model checker be more precise and find all false positives; on the

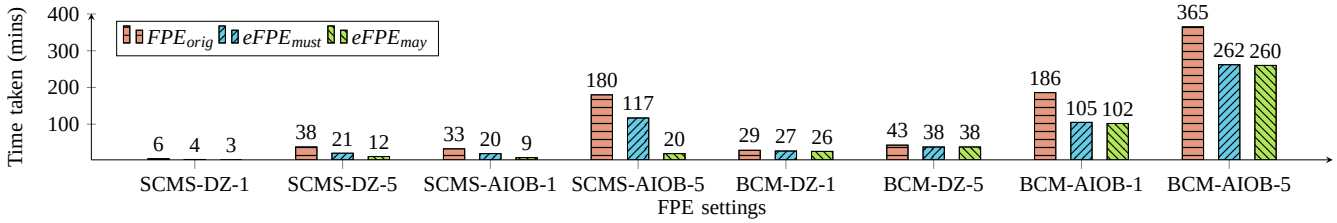


Fig. 7: False positives elimination time

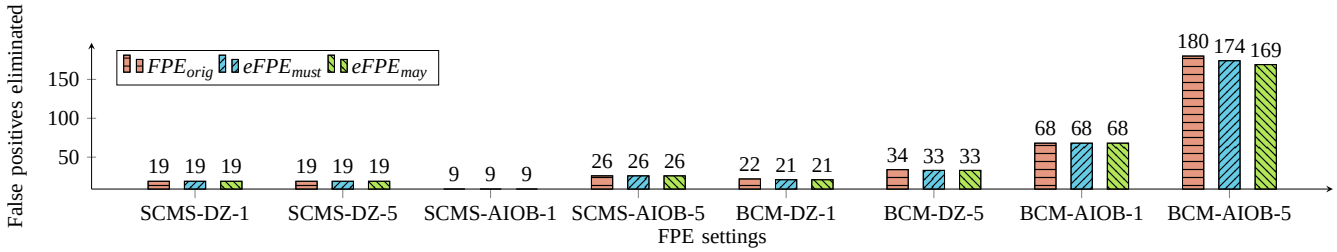


Fig. 8: False positives eliminated by FPE settings

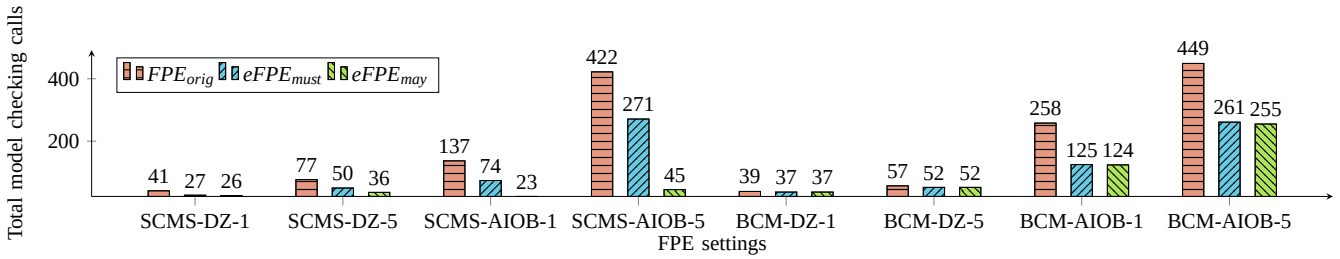


Fig. 9: Total model checking calls

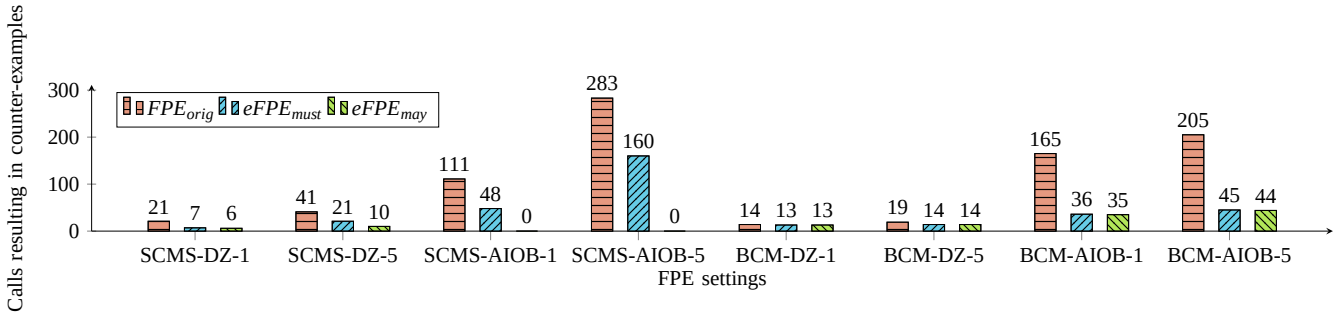


Fig. 10: Model checking calls resulting in counter-examples

other hand, we could let a few false positives slip through for efficiency of the process, but then we may need to spend time in their manual inspection.

2) May-*cnv Variables Vs* must-*cnv Variables:* Tables I and II indicate that, on SCMC application, the *may-cnv* variables identify significantly more number of RVCs as compared to the *must-cnv* variables while eliminating the same number of false positives that are eliminated by *must-cnv* variables. However, for BCM application *may-cnv* analysis identifies comparable number of RVCs although it fails to eliminate some (5) false positives that are eliminated by *must-cnv* variables. This explains the trade-off between speed-up and precision in false positives elimination when the *cnv* variables are computed at the different levels of over-approximation.

3) Constraining All Vs. Some Assertion Variables: Table II indicates that, requiring a single assertion variable to be a *cnv* variable identifies more RVCs as compared to the RVCs identified when all variables are required to be *cnv* variables. All false positives eliminated by ($eFPE_{must}$ and $eFPE_{may}$) are also eliminated by ($*eFPE_{must}$ and $*eFPE_{may}$), thus making the requirement of some assertion variable to be *cnv* variables a practical choice.

E. Generalization of Experimental Results

The experiments are performed using two applications and two properties to demonstrate the possibility of improving efficiency considerably with a negligible loss of precision in false positives elimination. It would be interesting to perform a wider range of experiments using a large set of benchmarks and verification properties to assert the behavior and effectiveness of the proposed apporach.

Even though the experiments are performed using proprietary static analysis tool (TCS ECA), the proposed approach can be applied on the warnings generated by any off-the-shelf tool. This is because the *cnv* variables used to identify a model checking call as a redundant are independant of the static analysis tool used. However, the gain in performance may vary depending on the static analysis tool and model checker in the combination because static analysis tools are known to have different false positive rate, and model checkers vary in their performance and scalability.

In the proposed approach, a warning is eliminated only when it is guaranteed to be a false positive. All the remaining warnings must be manually analyzed to ensure correctness at the warning points. Thus, the overall approach is sound and applicable to both crictical and non-critical systems alike.

V. Related Work

Broadly, static analysis and model checking have been combined in two ways to improve the precision of static analysis. In the first category, the information is iteratively exchanged between the two techniques [3], [8], [9]. For example, Brat et al. [3] and Junker et al. [9] have combined these two in such a way static analysis component iteratively exchanges information with the model checker. In the second category, static analysis and model checking are cascaded by using one after the other in a fixed order [6], [12], [14], [15], [17], [19]. In this approach, static analysis is used first and later model checking is used to eliminate false positives from the generated analysis warnings.

The cascading approach is more relevant for the comparison with our approach because it generates an assertion corresponding each warning reported by static analysis to be checked later by model checking. Wang et al. [19] have used such a combination to automatically detect code vulnerabilities. Further, Rödiger [15] has combined data flow analysis and model checking to improve the security vulnerability detection.

Post et al. [14] have adapted the approach to expand the calling context incrementally in order to deal with the scalability. In these combinations, calls to the model checkers are made without checking if they are redundant and the efficiency aspect is ignored. In our previous work [12], for the efficient elimination of false positives, we partitioned the generated assertions and verified only one representative assertion from each partition. However, it still requires making numerous calls to the model checker while verifying the representative assertion by incrementally expanding the calling context. To the best of our knowledge, only slicing [6], [19] and assertions partitioning [12] have been used for the efficiency purpose.

While we also follow a similar approach as used in the above combinations, we avoid making a model checking call directly by identifying if the corresponding verification can possibly result in futile counterexample. This technique, being orthogonal, can be combined with [6], [12], [14] to improve their performance.

VI. Conclusion and Future Work

Model checking of an assertion whose variables can take any arbitrary value is most likely to generate a counterexample. False positive elimination using the approach of context expansion inherently requires the input variables of a procedure to take arbitrary value, ironically generating spurious counterexamples in many cases. We have used these observations to reduce the number of the spurious counterexamples by proposing the concept of complete-range non-deterministic values (*cnv*) variables. The *cnv* variables are used to identify and avoid redundant verification calls, and thus improve performance during the model checking-based elimination of false positives.

In our experiments, we observed a large number of model checking calls resulting in counterexamples (around 58% of the total model checking calls). It is because often variables get their values from outside the calling context. We could avoid, on an average, 76% of these calls (recall) using the *cnv* variables and that with a precision of 97.3%. When the *cnv* variables are computed more precisely (*must* Vs *may*), the precision in identifying such calls increases but the recall decreases, indicating an interesting trade-off between the false positives eliminated and efficiency achieved. Our experiments demonstrate that this trade-off can be useful in practice to improve efficiency of false positives elimination considerably with a negligible loss in precision. Further, this trade-off can be possibly exploited at different levels by computing the *cnv* variables of varying precision. That is, *may-cnv* variables can be used when faster elimination of false positives is required while *must-cnv* variables are suited when higher precision is demanded.

Our empirical measurements show a significant gain (39.28%) in performance by avoiding 49.49% of the total model checking calls. This gain is achieved because the redundancy is computed in bulk using a light-weight static analysis whereas a model checker consumes a lot of time while verifying these redundant calls individually. However, the gain comes at the cost of missing elimination of some (2.78%) false positives. This is due to the inability to compute the *cnv* variables precisely, and is a downside of our appproach because each of the missed false positives needs to be manually inspected.

In near future, we plan to understand how the proposed approach behaves with a larger set of applications and verification properties by performing a wider range of experiments. Further, we aim to observe the effect of the precision of the commerical off-the shelf static analysis tools on the trade-off in false positives elimination.

Acknowledgement

We thank the anonymous reviewers for their thoughtful and detailed comments on the paper. We also thank Gabriella Carrozza for her help in shepherding the paper.

REFERENCES

[1] TCS Embedded Code Analyzer (TCS ECA). http://www.tcs.com/offerings/engineering_services/Pages/TCS-Embedded-Code-Analyzer.aspx.

[2] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[3] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 262–, Washington, DC, USA, 2001. IEEE Computer Society.

[4] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[5] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] P. Darke, M. Khanzode, A. Nair, U. Shrotri, and R. Venkatesh. Precise analysis of large industry code. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 306–309, 2012.

[7] D. Engler. Concur 2005 - concurrency theory. chapter Static analysis versus model checking for bug finding, pages 1–1. Springer-Verlag, London, UK, UK, 2005.

[8] A. Fehnker and R. Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. volume 9, pages 45–56. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2013.

[9] M. Junker, R. Huuck, A. Fehnker, and A. Knapp. Smt-based false positive elimination in static program analysis. In *ICFEM*, pages 316–331, 2012.

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

[11] A. D. Lucia. Program slicing: Methods and applications. In *SCAM*, pages 144–151, 2001.

[12] T. Muske, A. Datar, M. Khanzode, and K. Madhukar. Efficient elimination of false positives using bounded model checking. In *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*, pages 13–20, 2013.

[13] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[14] H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *ASE*, pages 188–197, 2008.

[15] W. Rödiger. *Merging Static Analysis and Model Checking for Improved Security Vulnerability Detection*. Masters, 2011.

[16] A. Tsitovich. Detection of security vulnerabilities using guided model checking. In M. Garcia de la Banda and E. Pontelli, editors, *Logic Programming*, volume 5366 of *Lecture Notes in Computer Science*, pages 822–823. Springer Berlin Heidelberg, 2008.

[17] M. Valdiviezo, C. Cifuentes, and P. Krishnan. A method for scalable and precise bug finding using program analysis and model checking. In J. Garrigue, editor, *Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 196–215. Springer International Publishing, 2014.

[18] K. Vorobyov and P. Krishnan. Comparing model checking and static program analysis: A case study in error detection approaches. In *International Workshop on Systems Software Verification (SSV'10)*, 2010.

[19] L. Wang, Q. Zhang, and P. Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *SCAM*, pages 165–173, 2008.

APPENDIX

FPE using Context Expansion Approach

The call to process an assertion A_n belonging to procedure f, during false positives elimination using calling context expansion, is invoked as $processAssertion(A_n, f, 0)$. The warning corresponding to the assertion A_n is eliminated only when this invocation returns *verificationSuccess*. The maximum allowed calling context length during context expansion is indicated by *maxCCL*.

Algorithm 1 False Positives Elimination using Calling Context Expansion

```
procedure PROCESSASSERTION(A_n, currFunc, level)
    result :=verifyAssertion(A_n, currFunc);
    if result ≠ "counterExample" then
        return result;
    level = level + 1;
    if level = maxCCL then
        return "Level_Limit_Reached";
    if currFunc is an application-entry function then
        return result;
    funcCallers = get callers of function currFunc;
    for caller ∈ funcCallers do
        result = runFPE(A_n, caller, level);
        if result ≠ "verificationSuccess" then
            return result;
    return "verificationSuccess";

procedure VERIFYASSERTION(A_n, currFunc)
    Verify A_n in the context of currFunc.
    if counter example is generated then
        return "counterExample";
    if A_n is verified successfully then
        return "verificationSuccess";
    if no decision is made in given time limit then
        return "timeOut";
```