On Implementational Variations in Static Analysis Tools

Tukaram Muske, Prasad Bokil
Tata Research Development and Design Center
54-B, Hadapsar I.E., Pune, India

***Abstract*—Static analysis tools are widely used in practice due to their ability to detect defects early in the software development life-cycle and that too while proving absence of defects of certain patterns. There exists a large number of such tools, and they are found to be varying depending on several tool characteristics like analysis techniques, programming languages supported, verification checks performed, scalability, and performance. Many studies about these tools and their variations, have been performed to improve the analysis results or figure out a better tool amongst a set of available static analysis tools. It is our observation that, in these studies only the aforementioned tool characteristics are considered and compared, and other implementational variations are usually ignored. In this paper, we study the implementational variations occurring among the static analysis tools, and experimentally demonstrate their impact on the tool characteristics and other analysis related attributes. The aim of this paper is twofold - a) to provide the studied implementational variations as choices, along with their pros and cons, to the designers or developers of static analysis tools, *and* b) to provide an educating material to the tool users so that the analysis results are better understood.**

Keywords*—*Static Analysis tools; Analysis Warnings; Implementational Variations.

I. Introduction

In the last decade and a half, use of static analysis tools for code quality assurance during development and maintenance of software systems has increased [1]. With just the code as input, these tools help automatically detect a class of errors like Divide by Zero (ZD), Overflow-Underflow (OFUF), and Array Index Out of Bound (AIOB). Given a program, a static analysis tool reports if any error (within classes of errors) is present in the program. The tool checks for safety of all the relevant program points, and classifies every point either as *safe* or *unsafe* (error). When such a classification cannot be made, a sound tool conservatively reports that program point as a warning. All such reported warnings need to be manually inspected [2].

Many open source and commercial static analysis tools are available, and they are of varied capabilities since they have been developed for different purposes [3], [4]. A lot of these tools have been studied and compared using several methods [3], [4], [5]. These studies include evaluating analysis tools based on one or more evaluation parameters such as analysis techniques, precision, error detection rate, scalability, performance, programming languages supported, and verification checks supported. We refer to these parameters as *characteristics* of the static analysis tools.

It is our observation that, besides the above characteristics, there exist variations in static analysis tools that arise due to different approaches taken while implementing the analysis and reporting phases. We refer to such variations as *implementational variations*. We find there is a need to study these variations as well since they directly or indirectly impact the tool characteristics. Such an impact is explained by using variation in the reporting of warnings from the code sample in Figure 1. In this code sample, there are 5 AIOB warnings (at lines 12, 32, 33, 42, and 43), and 2 OFUF warnings (at lines 15, and 18) since values of *a* and *b* are accepted at run-time. Verifying this code using some tools reports 3 AIOB warnings (at line 12, 32, and 43) and only one OFUF warning (at line 18), whereas some other tools report all the warnings (5 AIOB and 2 OFUF) for the same code.

We observe, the above change in the number of reported warnings is not due to the analysis techniques, but it is due to the different reporting styles as described in Section II-B. Thus, the reporting styles have direct impact on the precision of the tools, and comparing precision of two analysis tools by comparing warnings count alone may give a wrong impression. We intentionally avoid associating the explained variations with any of the actual tools since latest versions of the referred tools may contain changes contradicting the current discussion. Further, we limit our discussion to analysis warnings which correspond to run-time errors.

To the best of our knowledge, such implementational variations have not been identified or studied. In this paper, we identify 5 main approaches which in turn lead to 11 implementational variations observed in some of the commercial and open source static analysis tools. Further, we describe impact of these variations on the tool characteristics (like precision, performance, and scalability). We also study impact of these variations on other attributes of static analysis like -

1) understanding of the results by tool users. This is needed since many users report that the results of the static analysis tools are not easily comprehended [6].
2) number of verification cycles that one may need to perform after fixing the reported errors or warnings. This is an important analysis attribute that affects the end-to-end system verification time.

The key contributions of this paper are - a) identifying implementational variations found among various static analysis tools, *and* b) studying pros and cons of each implementational variation. The implementational variations are described in Section II, and their impact on tool characteristics

```
10. int a = readValue();
11. int b = readValue();
12. arr[b] = 0;

13. if (...){
14.   f1( );
15.   a++;
16. }
17. else{
18.   a++;
19.   f2();
20. }
```

```
30. void f1( )
31. {
32.   arr[a] =
33.    min(arr[b],100);
34. }

------------------------
40. void f2( )
41. {
42.   arr[b] =
43.    max(arr[a],-100);
44. }
```

Fig. 1: Example 1

```
   void f3( )
60.{
61. int a = 10;
62. if (...){
63.   a = 0;
64. }
65.
66. b1 = c1 / a;
67. if (...){
68.   b2 = c2 / a;
69. }
70.}
```

Fig. 2: Example 2

```
   #define SIZE 10
80.void f4( )
81.{
82. int b = 0;
83. int arr[SIZE];
84. if (...){
85.   arr[SIZE]=0;//error
86.   arr[c] = c++;
87.   b = nonDet( );
88. }
89. arr[b] = ...;
90.}
```

Fig. 3: Example 3

is empirically demonstrated in Section III. Sections IV and V, respectively, presents the related work and conclusion.

II. Implementational Variations

This section describes implementational variations among static analysis tools by classifying them into five variation categories that are observed during the source code analysis, or the results reporting phase, or both. Further, it identifies impact of these variations on characteristics of static analysis tools (precision, scalability, and performance), and few other attributes as described below:

1) *Results understandability* - how easy it is for a tool user to understand the results.
2) *Review efficiency* - how quickly a tool user is able to review all the generated warnings.
3) *Multiple verification cycles* (MVCs) - how many times the user runs the analysis tool to analyze the complete code by fixing all the reported errors/warnings.

Table I summarizes the described implementational variations and their impact. It includes -

1) *column 1:* variation categories, and examples with which these categories are explained.
2) *column 2:* implementational variations in each category.
3) *column 3:* errors/warnings reported on the corresponding example (mentioned in *column 1*), for each of the implementational variation. We use A_n (resp. O_n and Z_n) to indicate AIOB (resp. OFUF, and ZD) warning at line *n*. The prefix *err* to these notations is used to denote the respective error points at line *n*.

TABLE I: Summary of Implementational Variations

Variation category	Implementational variations	Example Results	Impacted tool characteristics	Impacted other analysis attributes
Range adjustment approach (Figure 1)	Range adjusted	A_{12}, A_{32}, A_{43}, O_{18}	precision↑	understandability↓ review efficiency↑ MVCs↑
	Range not adjusted	A_{12}, A_{32}, A_{33}, A_{42}, A_{43}, O_{15}, O_{18}	precision↓	understandability↑ review efficiency↓ MVCs↓
Warnings reporting approach (Figure 1)	All warnings reported	A_{12}, A_{32}, A_{33}, A_{42}, A_{43}, O_{15}, O_{18}	precision↓	understandability↑ review efficiency↓ MVCs↓
	First warning reported	A_{12}, A_{32}, A_{43}, O_{15}, O_{18}	precision↑	understandability↓ review efficiency↑ MVCs↑
	Group of warnings reported	(A_{12}, A_{33}, A_{42}), A_{32}, A_{43}, O_{15}, O_{18}	precision↑	understandability↑ review efficiency↑ MVCs↓
Error determination approach (Figure 2)	*All* paths error	Z_{66}, Z_{68}	detection rate↓	understandability↑
	Single path error	$errZ_{66}$, $errZ_{66}$	detection rate↑	understandability↓
Post-error paths analysis approach (Figure 3)	*Skip* post-error paths	$errA_{85}$	precision↑	understandability↓ MVCs↑
	Analyze post-error paths	$errA_{84}$, A_{86}, A_{89}, O_{86}	precision↓	understandability↑ MVCs↓
Support for property-wise verification	Available	-	scalability↑ analysis time↑	-
	Not available	-	scalability↓ analysis time↓	-

4) *column 4 and 5:* positively or negatively impacted analysis characteristics and other analysis attributes. We use ↑ and ↓ arrows, respectively, to indicate the positive and negative impact as compared to each other.

A. Range Adjustment Approach

The static analysis tools need to compute values of variables at a few or all program points, in order to decide safety of a relevant program point. For example in Figure 1, to check if the increment operation at line 15 can ever result in arithmetic overflow, a tool must identify the values that the variable *a* can take at the same program point. These tools are found to be based on a variety of analysis techniques like data flow analysis, abstract interpretation [7], and Difference-Bound Matrices [8] for computing values/ranges of the variables.

Precision of a static analysis tool, measured in terms of number of output warnings, is one of the most studied tool characteristics. We observe this characteristic is impacted by the approach with which values of the variables are computed. This approach (variation category) can be implemented on top of any of the analysis techniques resulting in two implementational variations based on - *whether the tools adjust the values/ranges of the variables after a reported warning point*. For example, some of the static analysis tools adjust range of *a* to [0..*arraySize*-1] after line 32 in Figure 1, where *arraySize* is size of the array. This is because, *a* is used as index of an array at line 32, and A_{32} is already marked as an AIOB warning. As a result of this adjusted range, the increment operation at line 15 is identified as a *safe* point. However, the tools that do not perform such range adjustment will report the same increment operation as a warning (O_{15}).

The impact of this variation category is summarized in Table I, and it indicates the presence (resp. absence) of range adjustment approach results in 4 (resp. 7) warnings for the code in Figure 1. The change in number of warnings illustrates impact of these variations on the tool's precision. Further, these variations also have impact on the -

1) understandability of the results, since many times the user faces difficulty in understanding warnings resulted after implementing range adjustment approach. For example, user may find it difficult to understand why the increment operation at line 18 is reported as a *warning* and the similar operation at line 15 is a *safe* point even though the values taken by *a* are same at both the program points.
2) review efficiency, since lower number of warnings gets generated when range is adjusted and lesser effort is incurred in their reviewing.
3) multiple verification cycles. This impact is discussed in [2]. We avoid describing it due to lack of space.

B. Warnings Reporting Approach

We observe there exists three different methods to report the warnings that are similar or correlated.

1) Report all warnings: This method reports all the warnings even if they are similar.

2) Report the first warning: In this method, only the first warning on a path is reported and reporting of the other similar warnings on the same path is skipped. Ensuring the reported (first) warning is not a defect, ensures all the rest warnings are also *safe*. However, in the case when the first one turns out to be an error, the required fix(es) may correct all the other similar warning points in one or more verification cycles [2].

3) Report group of similar/correlated warnings: In this method, all the similar warnings are reported as a group. This grouping is such that result of reviewing of a grouped warning (identified as leader warning [2], or dominant alarm [9]) is applicable to reviewing of all other grouped warnings.

The approach used to report the generated warnings also has a major impact on the precision of the static analysis tools, as the number of output warnings varies considerably as per the implemented reporting approach. This is explained using analysis results in Table I for the code sample in Figure 1. The first method leads to reporting of total 7 warnings (lower precision) while the other two methods will result in 5 warnings/groups (higher precision as compared to the first variation). Further, it is intuitive this variation category has direct impact on the results understandability, review efficiency, and MVCs.

C. Error Determination Approach

We observe the static analysis tools vary in two ways based on an approach to decide an error.

1) Error via all paths: The first way is to report an error only if it is an error through all the paths (*must error*), due to which less number of errors are reported (lower detection rate), but with higher accuracy. Further, in this approach, if a program point is an error via one or more paths, but not by all the paths, such a point is reported as a warning. For example, with this approach both the division operations in Figure 2 are reported as *divide by zero* warnings (Z_{66}, and Z_{68}).

2) Error via single path: The second way is to report an error if a program point is an error through any of the paths reaching at the error program point (*may error*). This way it reports a large number of errors, of which few may be spurious. For example, the division operations at line 8 and 10 will be reported as errors ($errZ_{66}$ and $errZ_{68}$), where $errZ_{68}$ may turn out as a false error.

D. Post-Error Paths Analysis Approach

This category deals with variation arising due to - *should the successive paths after a reported error be analyzed?* This question leads to two variations as described below.

1) Skip analysis of post-error paths: It is observed that, some tools skip analysis of the paths that originate from an error. These tools first demand corrective action for the reported error, and then only the later code portion is analyzed. For example, this approach stops analysis after reporting of $errA_{85}$ (Figure 3), and it results in skipping of analysis of the successive program points (points at line 86 and 87). Note that in this method only the program points that are always on erroneous paths will be skipped from the analysis. As a result of this method, the array access at line 89 is reported as *safe* even though non-deterministic values are assigned to *b* at line 87. It indicates this approach has a direct impact on the results understandability and also will surely lead to MVCs.

2) Analyze post-error paths: As opposed to the first approach, some tools take a conservative approach after reporting of an error and continue with the analysis. Thus, with this approach the program points that appear on the erroneous paths get analyzed. It is intuitive that this approach eliminates the MVCs and improves the results understandability. However, this approach results in more number of warnings (lesser precision) as compared to the warnings reported due to the first approach. For example, this approach reports 3 warnings (A_{86}, O_{86}, and A_{89}) for the code in Figure 3, while the first approach reports none of these.

E. Support for Property-wise Verification

It is our observation that, some tools allow selecting a property to be verified independent of other properties, while other tools analyze all the properties together providing no choice for selecting a property. The characteristics impacted by this approach are described below.

1) Scalability: The property-wise verification helps tools to achieve scalability on very large systems, since only the property-specific data and only the property relevant code need to be analyzed as against to the other approach. Also, many times a user is interested in focusing on a critical property like memory leaks, synchronization, or buffer overflows, and may wish to ignore other properties like AIOB, OFUF, and ZD.

2) Performance: The overall time taken to analyze the system can be very large in property-wise verification as compared to the analysis time with analyzing all properties together.

III. Experimental Results

We used TCS ECA [10] to implement the described implementational variations and demonstrate their impact on analysis characteristics. By implementing all these variations in one tool, we ensured the analysis technique and its computational precision remains same for all the implemented

variations. By default, TCS ECA does not adjust the ranges, reports warnings in groups, reports *all-path* errors, continues analysis of post-error paths, and allows selecting a property to be verified. We enabled one variation at a time while keeping other variations to default, so that only the impact of enabled variation is captured.

TABLE II: Experimental Results

Variation category	Implementational variations	#Warnings	#Errors	#VCs
Range adjustment	Range adjusted	865	6	3
	Range not adjusted	1266	6	2
Warnings reporting	All warnings reported	1381	6	2
	First warning reported	1280	6	3
	Groups of warnings reported	1266	6	2
Error determination	All paths error	1266	6	2
	Single path error	1251	21	2
Post-error paths analysis	Skip post-error paths	1153	3	4
	Analyze post-error paths	1266	6	2

We selected an embedded system application of 40 KLOC size, that was previously tested and verified. In order to check impact of the variation categories dealing with errors (as in Sections II-C, and II-D), we manually injected few faults in the application. Using each variant of TCS ECA, the buggy application was verified for AIOB, OFUF, and ZD properties since these are commonly checked properties in software verification. Table II presents results of this experiment in terms of number of output warnings and errors after first verification cycle. Column #VCs denotes the number of performed verification cycles. These results indicate impact of the implementational variations on tool's precision, error detection rate, and MVCs. In this table, results about the variation category - *support for property-wise verification* - are not provided since its impact on analysis time and scalability is obvious. It should be noted that the results are only to demonstrate the positive or negative impact of implementational variations, and it should not be used for quantifying the impact. The quantification would require a rigorous and wider range of experiments.

IV. Related Work

There exist several studies (like [3], [4], [5], [11], [12], [13]) focused on comparing the features of various static analysis tools. In these studies, precision and detection rates of analysis tools, are the two mostly compared characteristics. For example, Kratkiewicz [4] has used various parameters (detection rate, false alarm rate, performance, etc) to compare five static analysis tools that were capable of detecting buffer overflows in C code. Further, the number of verification properties and programming languages supported are widely compared features. For instance, Emanuelsson et al. [3] have compared properties supported by three commercial tools.

To the best of our knowledge, none of the existing studies have identified and studied the impact of implementational variations while comparing the characteristics of static analysis tools. Ignoring these implementational variations during studies that target comparing precision of the different analysis techniques, can give false impressions about the results, since these techniques can be coupled with different implementational variations. It is our first attempt to study the impact of the implementational variations on different analysis-related parameters and attributes.

V. Conclusion

In this paper, we have identified the variations observed during the implementation of analysis tools independent of their analysis technique, and have categorized them into five variation-categories. Each variation is described along with its impact-relation on analysis characteristics such as precision, error detection rate, and performance. Further, impact of these variations on other analysis related attributes (results understandability, review efficiency, MVCs) is also described.

We believe this study of implementational variations can be useful to the designers or developers of static analysis tools to make proper choices of implementational variations. Further, understanding of these variations can help tool users to understand the results better. In near future, we would like to work on quantifying impact of these variations on the studied analysis parameters.

References

[1] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 580–586.

[2] T. Muske, A. Baid, and T. Sanas, "Review efforts reduction by partitioning of static analysis warnings," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, Sept 2013, pp. 106–115.

[3] 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.

[4] K. J. Kratkiewicz, "Evaluating static analysis tools for detecting buffer overflows in c code," DTIC Document, Tech. Rep., 2005.

[5] V. Okun, A. Delaitre, and P. E. Black, "Report on the static analysis tool exposition (sate) iv," *NIST Special Publication*, vol. 500, p. 297, 2013.

[6] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 672–681.

[7] P. Cousot, "Abstract interpretation," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 324–328, 1996.

[8] A. Min, "A new numerical abstract domain based on difference-bound matrices," in *Programs as Data Objects*, ser. Lecture Notes in Computer Science, O. Danvy and A. Filinski, Eds. Springer Berlin Heidelberg, 2001, vol. 2053, pp. 155–172.

[9] D. Zhang, D. Jin, Y. Gong, and H. Zhang, "Diagnosis-oriented alarm correlations," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, vol. 1, Dec 2013, pp. 172–179.

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

[11] M. A. Al Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for java concurrency bugs," in *Third Swedish Workshop on Multi-Core Computing (MCC-10)*. Chalmers University of Technology, 2010.

[12] M. S. Ware and C. J. Fox, "Securing java code: Heuristics and an evaluation of static analysis tools," in *Proceedings of the 2008 Workshop on Static Analysis*, ser. SAW '08. New York, NY, USA: ACM, 2008, pp. 12–21.

[13] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*, Dec 2010, pp. 521–524.