VBSAC: A Value-Based Static Analyzer for C

Chi Li
School of Software, Tsinghua University
Beijing, China
chi-li18@mails.tsinghua.edu.cn

Min Zhou
School of Software, Tsinghua University
Beijing, China

Zuxing Gu*
School of Software, Tsinghua University
Beijing, China
guzuxing@sina.com

Guang Chen
School of Software, Tsinghua University
Beijing, China

Yuexing Wang
School of Software, Tsinghua University
Beijing, China

Jiecheng Wu
School of Software, Tsinghua University
Beijing, China

Ming Gu
School of Software, Tsinghua University
Beijing, China

ABSTRACT

Static analysis has long prevailed as a promising approach to detect program bugs at an early development process to increase software quality. However, such tools face great challenges to balance the false-positive rate and the false-negative rate in practical use. In this paper, we present VBSAC, a value-based static analyzer for C aiming to improve the precision and recall. In our tool, we employ a pluggable value-based analysis strategy. A memory skeleton recorder is designed to maintain the memory objects as a baseline. While traversing the control flow graph, diverse value-based plug-ins analyze the specific abstract domains and share program information to strengthen the computation. Simultaneously, checkers consume the corresponding analysis results to detect bugs. We also provide a user-friendly web interface to help users audit the bug detection results. Evaluation on two widely-used benchmarks shows that we perform better to state-of-the-art bug detection tools by finding 221∼339 more bugs and improving F-Score 9.88%∼40.32%.

CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

KEYWORDS

Static analysis, Value-based analysis, Bug detection

ACM Reference Format:
Chi Li, Min Zhou, Zuxing Gu, Guang Chen, Yuexing Wang, Jiecheng Wu, and Ming Gu. 2019. VBSAC: A Value-Based Static Analyzer for C. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3338998

1 INTRODUCTION

Modern software system development process is a complex and challenging task. Detecting and fixing software bugs is far cheaper earlier in the developing life cycle. To this end, static analysis has long prevailed as one of the most promising technique over the last years, which only requires the source code without having to run the code [10]. Unfortunately, static analysis tools face great challenges in practice [14]. One of the most important reasons is the high false positive rate. An empirical study [1] found that users would lose confidence in a tool if its false positive rate was higher than 30%. To improve precision, several tools only report the bugs with high confidence, which results in ignorance of potential problems.

Many researchers have devoted a lot of efforts to improve bug detection accuracy [5, 10]. However, the main obstacles to conducting a precise analysis are:

(1) **Insufficient value analysis.** Most of the existing static analysis techniques run their abstract analysis to reach a fixed-point before performing bug detection. But this can be problematic when running multiple analyses with different precision in a single module, where the precision of the tool will be fundamentally limited to the least precise analysis.
(2) **Inefficient information alternation.** Another widely-used technique performs bug-finding by separate checkers which maintain own abstract states. However, it may fail to detect bugs needed deep semantics in other checkers. For example, pointer analysis usually needs intervals to calculate an explicit offset. Therefore efficient information alternation matters very much in a precise static analysis.

In this paper, we present VBSAC, a value-based static analyzer for C code bug detection. In our tool, we employ a pluggable value-based analysis strategy, where analysis and detection processes are separated when traversing the control flow graph. With a memory skeleton recorder designed to depict memory objects generation and elimination as a baseline, different value-based plug-ins can be used for analyzing specific abstract domains and exchange program state information to strengthen the computation. Checkers will consume the analysis results to perform bug detection for different types. In this way, we can improve the precision and detect multiple types of bugs simultaneously without stopping when a bug is found.

[*] Correspondence author

ISSTA '19, July 15–19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338998

We generate concrete traces for each bug and filter the duplicated ones. The final bug reports are produced in a well-defined format for users to audit through a web interface. We implement our tool on top of LLVM-IR, which provides a well-suited low-level language and can be generated for different languages.

We evaluate our tool on test cases from two widely-used benchmarks covering eight Common Weakness Enumeration (CWE)[1]. The results of the experiments show that, with different plug-ins sharing program state information to refine the computation, VBSAC detects 221~339 more bugs and improves F-Score 9.88%~40.32% compared with state-of-the-art static analysis tools. Our tool, benchmark, and results can be downloaded from our website[2].

2 RELATED WORK

There are many research works [5] and widely-used tools [7, 9, 11, 12] applying static analysis to detect bugs in C code. CPAChecker [2] is a configurable tool aiming at integrating different program analysis and model checking approaches in one single formalism. Frama-C [8] provides static analysis embedded into a value analysis framework. Machine learning has been widely combined with traditional program analysis approaches to detect bugs in recent years [16]. However, these tools require users to manually annotate program properties or rely on the known bugs to train models, which is not as robust as an automatic approach. MPAnalyzer [6] is a tool to detect unintended inconsistencies in source codes. Same as MPAnalyzer focusing on a specific type of bugs, IntPTI [4] aims at integer errors, Vojdani [13] detects locking idioms in Linux device drivers, and Melton [15] is proposed for precise memory leak detection.

Main Difference VBSAC also takes advantage of well-known program analysis techniques to provide an automatic bug-finding tool for multiple types of bugs. Different from the existing static analysis techniques which run their abstract analysis before performing bug detection [2] or find bugs in separate checkers maintaining own abstract states [7, 11]. We employ a pluggable value-based strategy, where different plug-ins maintain its own abstract domain values and exchange program properties and refine values to improve the precision.

3 DESIGN OF VBSAC

As presented in Figure 1, the work-flow of VBSAC consists of three phases. Code parsing phase preprocesses the source code into LLVM-IR and builds the control flow automata(CFA) which is an extended control flow graph. Bug detecting phase constructs the analysis module with different value-based plug-ins and conducts the bug-finding process. When detecting phase finished, report generating phase will eliminate duplicated bug traces and produce the final bug report. End users can review the results via the web interface.

3.1 Code Parsing Phase

The use of our tool is to provide LLVM-IR files as input. For a single file, we can use clang to preprocess the source file. For projects, the compilation commands in Makefile can add -E flag to preprocess all the *.c files to *.i files, which are self-contained source files with macros expanded and necessary declarations included. To provide a precise analysis, we build CFA by extending the traditional control flow graph with edges classified into two types, where *ControlEdge* is proposed to carry the LLVM-IR instruction to conduct the concrete semantic computation and *SummaryEdge* provides a summary mechanism to support analysis for large-scale code by skipping the loops and function calls using the summaries pre-computed.

3.2 Bug Detecting Phase

To improve the accuracy of our tool, we employ a pluggable value-based analysis strategy, where *Analysis Module* maintains the correct program properties in each domain separately and *Checker Module* detects bugs. In this way, we can improve the precision and detect multiple types of bugs simultaneously without stopping the analysis when a bug detected.

Analysis Module is designed to conduct abstract analysis on the input CFA. To provide a flow-sensitive and interprocedural analysis, we implement three fundamental analysis, *Location, Call-Stack and Bound* to record the program location, call depth and loop iteration while traversing the CFA. We explicitly evaluate the value of path conditions to provide a path-sensitive analysis. We use a *Memory Skeleton Recorder(MSR)* to maintain the memory objects generated and destroyed. With *MSR* as a baseline, a value plug-in maintains own abstract state and exchanges information to refine the computation results with others. Such as pointer plug-in will use range plug-in to explicitly calculate the offset. In the analysis process, we maintain an abstract program state as $S =< m, P >$, where m is the memory structure computed by *MSR* and P consists of different value plug-ins such as pointer and interval. When the analyzing phase traverses the CFA edge as e, we compute the new state in two steps: $S(m, P) \xrightarrow{e} S^{'}(m', P)$ to update the memory m and record the changes as Δ, and $S^{'}(m', P)) \xrightarrow{e,\Delta} S^{''}(m', P')$ for each plug-in $p \in P$ to update its own abstract state.

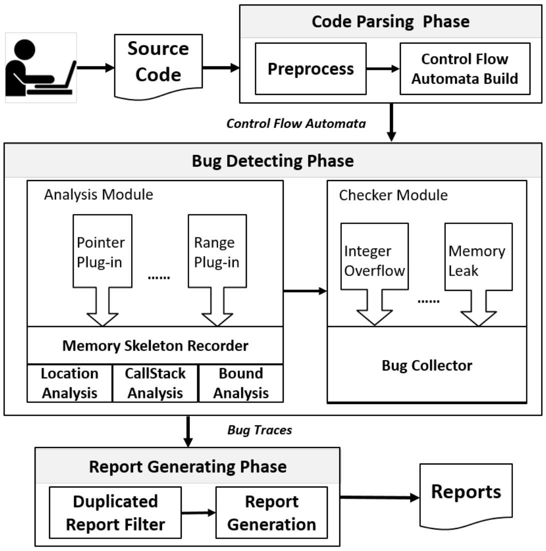


Figure 1: The architecture of VBSAC.

[1] https://cwe.mitre.org
[2] https://github.com/tomgu1991/VBSAC

Checker Module is designed to separate bug detection from the abstract program state computation for easy configuration and extension. Each checker will detect a specific type of bugs according to the abstract program state. We detect bugs in two steps: *Pre-Edge* which is before the program state updated and *Post-Edge* for the updated state. *Pre-Edge* will check the program properties according to the state S, where bugs may be triggered in the current instruction. *Post-Edge* will check the program properties according to the new state S''. This step is designed to find the bugs requiring the updated states. Another benefit of the separated *Checker Module* is that our analysis will not stop when a bug is detected, for the *Analysis Module* will maintain the correct program properties. We generate the concrete traces for each bug and use the *Bug Collector* to gather all the bug traces.

3.3 Report Generating Phase

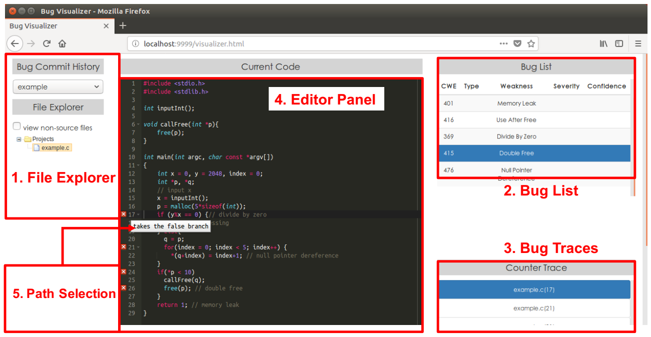
To provide a precise static analysis, we expand loops and function calls according to the Bound Analysis. In this way, a bug site may be triggered several times, such as a null pointer dereference problem in a loop. To eliminate this kind of duplicated bugs, we use a bug filter to remove the same reports. But, we will remain the bugs from a different context. For example, a bug in a function f, which is called from g_1, g_2, g_3. We will produce three bug traces to these bugs, for they are from different call-context. For all the bugs detected by VBSAC, we generate the traces for each of them with the detailed information into an XML file, such as file name and line number. The XML file can be used in the Web interface for users to audit the results.

4 IMPLEMENTATION

VBSAC is built on top of LLVM-IR 3.9, which provides a typed, static single assignment(SSA) and well-suited low-level languages. LLVM-IR can be generated by compiler front-ends for different languages, indicating tools can be extended to multiple languages. We implement a Java API for LLVM-IR based on *javacpp*[3].

For bug detecting phase, we implement our analysis extending CPA algorithm [2]. We have implemented Location Analysis, Call-Stack Analysis, and Bound Analysis as fundamental components to traverse the control flow automata. Currently, we implement a pointer plug-in based on AccessPath [3] and an interval plug-in based on multi-range which extended to maintain a fix-length list of traditional range in Analysis Module. Checkers are implemented to detect bugs relevant to integer and pointer, such as integer overflow, divide by zero, memory leak and so on.

To help users better understand our detection results, we provide a web interface to review the potential bugs as shown in Figure 2. The file explorer lists source files of the current bug-finding task. All the bugs are listed on right top of the page and sorted according to CWEID. By selecting a concrete bug, traces to trigger it are listed in the right bottom panel with file names and line numbers. In the Editor Panel, we show the source code with the traces with a red cross at the beginning, where semantic information will be presented when cursor rolls over.



```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int inputInt();
5
6  void callFree(int *p){
7      free(p);
8  }
9
10 int main(int argc, char const *argv[])
11 {
12     int x = 0, y = 2048, index = 0;
13     int *p, *q;
14     // input x
15     x = inputInt();
16     p = malloc(5*sizeof(int));
17     if (y%x == 0) {// divide by zero
20         q = p;
21         for(index = 0; index < 5; index++) {
22           *(q+index) = index+1; // null pointer dereference
23         }
24     if(*p < 10)
25         callFree(q);
26         free(p); // double free
27     }
28     return 1; // memory leak
29 }
```

CWE	Type	Weakness	Severity	Confidence
401		Memory Leak		
416		Use After Free		
369		Divide By Zero		
415		Double Free		
476		Null Pointer Dereference		

example.c(17)

example.c(21)

Figure 2: The web interface of VBSAC.

5 EVALUATION

Experiment Setup. To evaluate VBSAC, we compare with three open-source tools(Cppcheck[11], Clang Static Analyzer [7], Infer[12]) and a commercial tool with academic permission(PVS-Studio[9]). They are chosen because of widely mentioned in static analysis publications. Besides, they are fully automatic tools without manual intervention to add assertions or write specifications, which is needed in tools such as CPAChecker [2] and Frama-C [8]. Bug detection accuracy is measured by precision, recall and F-Score which is the harmonic mean of precision and recall.

$$precision = \frac{number\ of\ correct\ bug\ reports}{number\ of\ all\ the\ reports} \tag{1}$$

$$recall = \frac{number\ of\ correct\ bug\ reports}{number\ of\ ground\ truth} \tag{2}$$

$$F\text{-}Score = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

The benchmark consists of 568 bugs specific to integer and pointer from two widely-used benchmarks: Juliet Test Suite for C/C++ V1.3[4] and ITC-benchmark[5]. Test cases are classified according to CWEID as shown in the first two columns in Table 1, including 190-integer-overflow, 191-integer-underflow, 369-divide-by-zero, 401-memory-leak, 415-double-free, 416-use-after-free, 457-use-of-uninitialized-variable and 476-null-pointer-dereference. Each case is composed of a *bad* part with a bug and a *good* part without the bug for precision and recall evaluation. All the experiments are conducted on a desktop under Ubuntu 16.04 with 4G memory.

Results. We present the result of each tool in two columns in Table 1, including the number of all the reports as *#Rept*(*true positives* + *false positives*), the number of the correct bugs in the reports as *#Correct*(*true positives*). We use - to present that a tool does not support this type of bugs. In the last four rows, we list the analysis time, precision, recall and F-score of each tool.

Columns 3-4 illustrate the result of Infer, which is developed by Facebook for pointer and memory analysis. Infer successfully detects 217 true bugs in the benchmark with a high precision of 92.34%. However, it misses 49 bugs, resulting in a lower recall of 78.62%. After manually analyzing the reports from Infer, we find it

[3] https://github.com/bytedeco/javacpp

[4] https://samate.nist.gov/SRD/testsuite.php

[5] https://github.com/regehr/itc-benchmarks

Table 1: Evaluation Result of VBSAC

General Info		**Infer**		**Cppcheck**		**Clang-SA**		**PVS-Studio**		**VBSAC**	
CWEID	*#Bug*	*#Rept*	*#Correct*	*#Rept*	*#Correct*	*#Rept*	*#Correct*	*#Rept*	*#Correct*	*#Rept*	*#Correct*
190	80	-	-	23	23	13	13	18	16	85	79
191	59	-	-	6	6	1	1	40	30	64	58
369	65	-	-	9	9	12	12	11	11	65	65
401	81	82	66	22	20	78	71	92	62	84	80
415	62	62	62	60	58	59	59	55	55	64	62
416	50	23	23	0	0	6	6	58	46	54	50
457	88	-	-	53	53	69	67	66	63	102	85
476	83	68	66	46	45	54	53	62	59	82	74
Summary	568	235	217	219	214	292	282	402	342	600	553
Time(s)		116.6		**30.7**		226.6		189.1		655.8	
Precision(%)		92.34		**97.71**		96.57		85.07		92.17	
Recall(%)		78.62		37.67		49.65		60.21		**97.40**	
F-score(%)		84.92		54.38		65.58		70.52		**94.70**	

#Bug is the number of ground truth. *#Rept* and *#Correct* is the number of bug reports and the correct bugs in the reports.

fails to detect the bugs requiring a path-sensitive and interprocedural analysis, resulting in missing bugs cross functions and infeasible paths to be considered. In VBSAC, we employ an interprocedural analysis and combine interval and pointer to explicitly evaluate a path condition to improve the analysis accuracy. VBSAC detect 266=(80+62+50+74) true bugs with 18 false positives for the tracks supported by Infer, indicating that VBSAC improves the recall from 78.62 to 93.66%.

From columns 5-10, we can observe that Cppcheck and Clang-SA perform better in precision with a ratio of 97.71% and 96.57%. However, they miss more than half of the bugs in the benchmark, especially in interval relevant bugs. After analyzing the results of these two tools, we find they preferred a conservative strategy that they only report the bugs with high confidence. PVS-Studio detects 10.56%∼22.54% more bugs compared with the above tools with a moderate loss of precision. However, we find that all of them fail to provide a precise value analysis, especially for interval analysis. This compromising strategy can speed up the analysis process, while it has high impacts on the loop unrolling and path selection, resulting in potential bugs ignored because of lack of possible range of values. With different value-based plug-ins maintaining own domains and exchanging properties, VBSAC provides a more accurate analysis solution. Compared with all the tools on F-score, VBSAC achieves 94.70% on the benchmark, improving 9.88%∼40.32% with the capability of multiple types of bugs.

Therefore, it is reasonable to conclude the combination of different value-based plug-ins will help VBSAC to perform a more precise analysis and detect more types of bugs. With the bug traces displayed in web interface, we can understand the bug details better, which will promote the fixing process.

6 CONCLUSION

In this paper, we present VBSAC, an automatic static analyzer for C. It employs a pluggable value-based strategy to provide a flow-sensitive, path-sensitive, and interprocedural analysis. We illustrate the tool architecture and implementation. By providing a web visualizer, we hope to fill the gap between end users and the tool. The evaluation on two widely-used benchmarks demonstrates that VBSAC performs better on precision and recall compared with the state-of-the-art tools. In the future, we plan to implement more plug-ins and adapt our tool to more real-world projects.

REFERENCES

[1] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[2] Dirk Beyer and M. Erkan Keremoglu. 2009. CPAchecker: A Tool for Configurable Software Verification. *CoRR* abs/0902.0019 (2009).

[3] Ben-Chung Cheng and Wen-mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI 2000, Canada.* 57–69.

[4] Xi Cheng, Min Zhou, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2017. IntPTI: automatic integer error repair with proper-type inference. In *ASE 2017, Urbana, IL, USA.* 996–1001.

[5] Aurélien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. 2015. Evaluating Bug Finders - Test and Measurement of Static Code Analyzers. In *COUFLESS 2015, Florence, Italy.* 14–20.

[6] Yoshiki Higo and Shinji Kusumoto. 2014. MPAnalyzer: a tool for finding unintended inconsistencies in program source code. In *ASE 2014, Sweden.* 843–846.

[7] Clang Static Analyzer. Accessed on May 28. 2018. http://clang-analyzer.llvm.org/

[8] Frama-C. Accessed on May 28. 2018. http://frama-c.com/

[9] PVS-Studio. Accessed on May 28. 2018. https://www.viva64.com/en/pvs-studio/

[10] Ganesh Selvaraj, Gerald Weber, and Christof Lutteroth. 2017. Efficient Program Analyses Using Deductive and Semantic Methodologies. *2017 IEEE 13th International Conference on e-Science (e-Science)* (2017), 440–441.

[11] Cppcheck: A tool for static C/C++ code analysis. Accessed on May 28. 2018. http://cppcheck.sourceforge.net/

[12] Infer: A tool to detect bugs in Java and C/C++/Objective-C code. Accessed on May 28. 2018. https://fbinfer.com/

[13] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *ASE 2016, Singapore.* 391–402.

[14] Jim Witschey, Olga A. Zielinska, Allaire K. Welk, Emerson R. Murphy-Hill, Christopher B. Mayhorn, and Thomas Zimmermann. 2015. Quantifying developers' adoption of security tools. In *ESEC/FSE 2015, Bergamo, Italy.* 260–271.

[15] Zhenbo Xu, Jian Zhang, and Zhongxing Xu. 2015. Melton: a practical and precise memory leak detection tool for C programs. *Frontiers Comput. Sci.* (2015), 34–54.

[16] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2017. Machine-Learning-Guided Typestate Analysis for Static Use-After-Free Detection. In *Computer Security Applications Conference.* 42–54.