CoBOT: Static C/C++ Bug Detection in the Presence of Incomplete Code

Qing Gao[1,2,3†], Sen Ma[1,2†], Sihao Shao[1,2†], Yulei Sui[4‡], Guoliang Zhao[1,2,5‡],
Luyao Ma[1,2], Xiao Ma[1,2], Fuyao Duan[1,2], Xiao Deng[1,2,3], Shikun Zhang[1,2], Xianglong Chen[6]
[1]National Engineering Research Center for Software Engineering, Peking University
[2]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[3]School of Electrical Engineering and Computer Science, Peking University
{gaoqing,masen,shaosihao,maluyao,maxiao94,duanfuy,dxeecs,zhangsk}@pku.edu.cn
[4]CAI and School of Software, University of Technology Sydney, Australia, yulei.sui@uts.edu.au
[5]CASIC_CQC Software Testing and Assessment Technology (Beijing) Corporation, Ltd., zhaogl@sqa-bj.com
[6]CASC Software Testing Center, 1042399133@qq.com

ABSTRACT

To obtain precise and sound results, most of existing static analyzers require whole program analysis with complete source code. However, in reality, the source code of an application always interacts with many third-party libraries, which are often not easily accessible to static analyzers. Worse still, more than 30% of legacy projects [1] cannot be compiled easily due to complicated configuration environments (e.g., third-party libraries, compiler options and macros), making ideal "whole-program analysis" unavailable in practice. This paper presents CoBOT [2], a static analysis tool that can detect bugs in the presence of incomplete code. It analyzes function APIs unavailable in application code by either using function summarization or automatically downloading and analyzing the corresponding library code as inferred from the application code and its configuration files. The experiments show that CoBOT is not only easy to use, but also effective in detecting bugs in real-world programs with incomplete code. Our demonstration video is at: https://youtu.be/bhjJp3e7LPM.

KEYWORDS

incomplete code, static analysis, bug detection

ACM Reference Format:
Qing Gao, Sen Ma, Sihao Shao, Yulei Sui, Guoliang Zhao, Luyao Ma, Xiao Ma, Fuyao Duan, Xiao Deng, Shikun Zhang, Xianglong Chen. 2018. CoBOT: Static C/C++ Bug Detection in the Presence of Incomplete Code. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3196321.3196367

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05.
https://doi.org/10.1145/3196321.3196367

1 INTRODUCTION

Static analysis tools, such as Coverity [3], Klocwork [4], and Clang [5], are widely used to find bugs in the early stage of the whole software development cycle. To increase the accuracy of static analysis, most tools require "whole program analysis" (1) Complete source code of the target program needs to be available, including not only the application code but also library code such as Std, VC and QT. (2) The program needs to be compiled successfully by providing detectors with necessary configuration information, such as the exact locations of libraries' header files, and macros used in the compilation environment. However, it is often a difficult and time-consuming task for users of static analysis tools to manually configure the compilation environment for a successful compilation, especially, for correctly configuring large-scale C/C++ programs with multiple program modules interacting with many third-party libraries. In 2017, CQC [1](a famous company for software testing in China) surveyed test engineers from more than 50 quality assurance (QA) departments in different companies, and found that more than 30% of C/C++ projects cannot be easily compiled due to the complexity of building a proper environment. Therefore, they are not analyzable by most of the static analysis tools. The reasons are as follows.

First, large code repositories often contain legacy code with complicated configurations that refer to various third-party libraries. Even worse, users of static analysis tools often lack knowledge in project configuration to include necessary and correct files for whole-program analysis, especially when the project has sophisticated third-party library dependence. Second, users of static analysis tools are often unable to configure libraries properly, even though they are aware of the importance of analyzing the libraries due to complicated configurations for legacy code. The complicated configurations come from two aspects: a) Library compatibility. For example, the code needs QT8.5 third party library while the

[†]Co-First authors.
[‡]Co-Corresponding authors.
This work was supported by the National Key Research and Development Program of China (2017YFB0802900), Beijing Natural Science Foundation (4182024), the China Postdoctoral Science Foundation (2017M620524), and Australian Research Council grant DE170101081.

environment only has QT7.0, which is incompatible with QT8.5. b) Lots of manual efforts. Test engineers often complain about the existence of many legacy programs that make it tedious and time-consuming to compile the programs [6]. Furthermore, the source code of a project sent to static analysis tools may be incomplete or contains compilation errors,, which makes whole-program analysis extremely difficult.

To address these issues, we present a static analysis tool, called CoBOT, that supports bug detection for C/C++ programs with good accuracy even if the code is incomplete or cannot be compiled due to configuration problems. CoBOT performs automatic library matching by downloading its corresponding versions inferred from the application code and its configuration files. We apply six heuristic strategies for systematically resolving the missing libraries of an application, thereby providing precise and sound results to improve the accuracy of a wide variety of bug detectors, including detecting null pointer dereferences, use of uninitialized variables, memory leaks and use-after-frees. Through our experiment, we demonstrate that CoBOT effectively found C/C++ program bugs statically in the presence of incomplete code, with comparable accuracy when the source code is complete.

2 ARCHITECTURE

The overall CoBOT architecture is shown in Fig. 1, which is distributed across three phases: parsing, analysis, and detection. In these phases, there are six highlighted key strategies to achieve precise results for analyzing real-world projects in the presence of incomplete code: 1) Library matching: matching library due to different versions of a third-party library; 2) Header file retrieval: performing fine-grained header file matching due to header files with the same name (files residing in different program modules); 3) Macro replacement: retrieving macro values without compilation; 4) Binding building: building links between function/variable declarations and their uses; 5) Function summary matching: matching library functions in the absence of their definitions; and (6) Heuristic defect detection: adopting different detection methods according to different types of defects.

2.1 Parsing Phase

CoBOT uses Eclipse CDT plugin [7] to perform lexical and syntax analysis and generate enhanced abstract syntax trees (ASTs) for C/C++ projects. However, without the whole program, the ASTs will be incomplete, affecting the accuracy of static analysis. We propose the following four strategies to build enhanced ASTs for incomplete code, thereby providing necessary auxiliary information to improve precision and soundness of static analysis.

(1) Library Matching. Once a program is uploaded into CoBOT, CoBOT retrieves the proper library automatically under two scenarios. First, CoBOT matches the corresponding libraries based on the configuration files from the target project. For example, projects created in Visual Studio have configuration files with the suffix "vcxproj" or "vcproj". CoBOT analyzes such configuration files, to acquire the detailed version based on its XML description. Second, if third-party libraries can not be determined by the configuration files, CoBOT chooses a corresponding library based on the calculated similarity score by matching the included headers in the target project with the headers in different libraries. Supposing that the number of supported libraries by CoBOT is N, and the set of library files is denoted as L_i for Library i, the set of special headers Sp_i for Library i is calculated as follows:

$$Sp_i = \{fn | fn \in L_i, fn \notin L_m, m \neq i, 0 \leqslant i, m < N, i, m \in \mathbb{Z}\}$$

Note that CoBOT only considers file names and ignores the file paths in this strategy. Then CoBOT matches included header file names in the program with Sp_i for Library i. The similarity score is calculated using the following formula:

$$Score_i = \frac{Number\ of\ matched\ special\ headers\ in\ Sp_i}{Number\ of\ total\ included\ headers\ of\ the\ project}$$

CoBOT chooses the library with the highest score. In the case of a tie, CoBOT will choose one among the candidate libraries randomly, and then automatically download library files. CoBOT now supports the following libraries: vc6.0, vs08, vs10, vs12, MinGW, gcc, tornado, QT4 and QT5.

(2) Header File Retrieval. In a C/C++ project, multiple headers residing different modules of a third-party library may have the same name. Matching the wrong one may affect the correctness of static analysis. CoBOT supports not only library matching, but also header file matching in a fine-grained way. To match a header file *header* included by a source file *src*, CoBOT searches through the identified library and the directory of application code of the project, using the included header name or path suffix, such as "Windows.h" or "sys/config.h". If there are multiple header files matched, we will analyze the most appropriate one. Supposing that the path string of the ith candidate header is: $String_i = h_{i,1}/h_{i,2}/.../h_{i,j}$, the path string of the source file *src* is: $String_{src} = s_1/s_2/.../s_k$, where "/" represents the path separator of different operating systems, CoBOT calculates the distance $dist_i$ between each candidate header and *src* using the following formula:

$$dist_i = \sum_{level=1}^{max(j,k))} (edit\_distance(h_{i,level}, s_{level}) * 2^{max(j,k)-level})$$

The function *edit_distance* represents the Levenshtein distance. The intuition of this formula is that the candidate header file with the most similar path prefix, rather than suffix, should be considered with highest priority. In other words, we choose the most "close" header file based on the source file that includes the header file. Finally, CoBOT sorts the candidate files H_i and H_j:

$$H_i < H_j \textbf{ iff } dist_i < dist_j$$

The first candiate file is chosen. If no candidate files are found, CoBOT provides missing file information to users.

(3) Macro Replacement. In C/C++, macros are widely used to produce different versions of a project. When compiling a program, the values of macros defined by the compiler or by the program can be obtained automatically. However, without whole-program information, these values are incomplete. To tackle this problem, our insights are: (1) Definitions of macros are usually provided in configuration files (e.g., Makefile), which address the macro-missing problem; (2) In

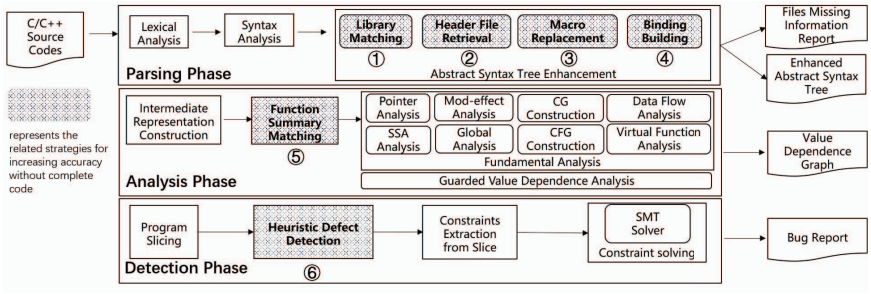

Figure 1: The overall architecture of CoBOT.

Strategy 1, CoBOT has identified the corresponding library, and we can then load the values of a macro defined in the library, which address the compiler macro-missing problem. Hence, CoBOT resolves the missing macro definitions, so that they can be used in the following parser strategies for macro replacement to improve accuracy.

(4) Binding Building. The links between variable uses (or function calls) and their definitions are known as binding, which is crucial for inter-procedural analysis. To build missing bindings without the whole program, CoBOT searches variable/function definitions of the same name, and compares the path similarities between the files that define a variable/function and those that use the variable/function, following the same formula in the second strategy.

2.2 Analysis Phase

CoBOT transforms ASTs to a program-dependence model, called Value Dependence Graph (VDG) [8], similar to SVFG [9] for defect detection. First, CoBOT builds summaries for available library functions. Then CoBOT performs a series of analyses including control-flow graph (CFG) construction, call graph (CG) construction, virtual function analysis [10], SSA analysis [11], pointer analysis [12], modification side-effect analysis [13], and data flow analysis [14].

(5) Function Summary Matching. Function summaries are constructed by intra-procedural analysis and used in the inter-procedural analysis phase [15]. Our strategy is to match summaries for library functions invoked in application code. If the callee of a function call can not be found in the application code or any of the available third-party libraries, CoBOT will retrieve it in the function-summary database. CoBOT calculates the Levenshtein distance between current function's name and that in the database, to determine the library function that has the shortest length. For example, if a function name is called x_malloc, CoBOT will match *malloc*. If none of the summaries can be matched, CoBOT will finally take it as a user-defined function, handled in Strategy 6.

2.3 Detection Phase

In the last phase, CoBOT detects bugs based on the VDG. First, CoBOT slices VDG according to different defect patterns. Second, it generates defect constraints on each slice. Finally, CoBOT solves these constraints by SMTInterpol [16] to determine if the slice contains a bug. When the whole program code is incomplete, function definitions may be missing, which may cause static bug detectors to report false positives or negatives. To address this problem, CoBOT uses different strategies based on different defect patterns.

(6) Heuristic Defect Detection. In this strategy, CoBOT mainly analyzes user-defined call sites which do not have corresponding definitions. Handling such call sites depends on the type of a defect pattern. For example, supposing that the checker detects use of uninitialized variables, the slice of the defect is "$int\ var;\ fun2(\&var);\ int\ b = var;$", and the call site "$fun2(\&var)$" is not parsed or matched successfully in the previous strategies. CoBOT conservatively assumes that var is initialized in $fun2$ to reduce false positives.

3 EVALUATION

We evaluated the effectiveness of our approach in two experiments: (1) comparing the bug detection results by applying and not applying the six heuristic strategies of CoBOT, and (2) comparing the analysis results by compiling and not compiling the projects. If the projects were compiled, the compiling information (e.g., included header file paths) is provided to CoBOT, otherwise, CoBOT builds enhanced ASTs for analyzing the incomplete program.

Benchmark: We evaluated the performance and accuracy by using a set of real-world C/C++ projects, which are open-source projects as shown in Table 1. We chose null pointer dereferences and use of uninitialized variables as two major clients, because the accuracy of detecting these two types of bugs is particularly affected by compilation environments and correctly analyzing third-party libraries. All of our experiments were conducted on a machine with an 8-core i7-4790 3.6GHz CPU, and 8GB RAM.

Experiment 1: CoBOT has a switch to turn on and off the six heuristic strategies. All the projects were parsed into enhanced ASTs for analysis using Eclipse CDT plugins without compiling the whole program, and the results are shown in Table 1, in which "Relative False Negative Rate" is calculated by using the formula $(1 - num(Bugs)/num(TotalBugs))$, where $TotalBugs$ is the total bugs reported by all methods. We turned on the switch to apply the strategies, and the false positive (FP) rate was 5.31%, while the relative false negative (FN) rate was 18.31%. Then we turned off the switch, and the FP rate was 90.41%, while the relative FN rate was 69.01%. First, using the strategies, the FP rate decreased, because CoBOT(on) dealt with library calls with function summaries. For instance, for $scanf(\&v, \text{"\%d"})$, CoBOT(on) considers v to be initialized by checking the function name against the library summaries. Second, the relative false negatives

Table 1: Detection results of null pointer dereferences and use of uninitialized variables.

Benchmark	LOC	CoBOT(off)		CoBOT(on)		CoBOT(whole)	
		B/R	Time	B/R	Time	B/R	Time
bzip2-1.0.6	8117	0/3	32s	0/0	27s	0/0	23s
zlib-1.2.8	33346	3/15	57s	5/5	58s	0/0	35s
tommyds-2.1	36750	5/25	1min21s	18/19	1min18s	18/19	1min4s
VS_LIBEMU(scdbg)-0.2.0	55727	7/207	56s	30/30	1min2s	26/26	57s
ipimitool-1.8.15	79760	6/138	1min1s	12/13	2min34s	10/11	2min20s
mujs-1.0.1	15831	0/5	21s	2/5	41s	2/5	35s
Sjeng-Free-11.2	18035	13/25	32s	17/17	42s	16/16	48s
link41b(parser)-4.1	22372	2/3	28s	3/5	36s	3/4	27s
gobmk(gnugo-3.8)	87575	0/88	2min13s	14/14	2min7s	16/16	4min5s
wrk-4.0.0	77089	3/26	2min14s	6/9	1min48s	6/6	2min16s
libnl-1.0	81776	8/44	1min55s	15/15	53s	14/14	53s
postgres-x2-1.2.1	959200	41/339	17min32s	110/113	17min26s	92/94	13min16s
Total Bugs/Reports		88/918		232/245		203/211	
False Positive Rate%		90.41		5.31		3.79	
Relative False Negative Rate%		69.01		18.31		15.77	

"B/R" is "Bugs/Reports". "Bugs" is the number of true bugs. "Reports" is the total number of reported bugs. "CoBOT (off)" represents the approach without the whole program and without applying the six heuristic strategies. "CoBOT (on)" represents the approach without the whole program but applying the heuristic strategies. "CoBOT (whole)" represents the approach with the whole program and compilation.

also decreased significantly using the six strategies, because CoBOT(on) successfully matched function call sites with their definitions, using our library summaries. For instance, in the code "$int * \ p = null;\ isoc99\_strlen(p);$", the declaration of $isoc99\_strlen$ could not be bound, and CoBOT(on) matched it heuristically with the library function *strlen*. Library summaries of CoBOT(on) indicated that the first parameter of *strlen* is dereferenced. Hence a null pointer dereference was reported.

Experiment 2: As shown in Table 1, by compiling the whole program, CoBOT(whole) kept the FP rate as 3.79%, and the relative FN rate as 15.77%. Compared to CoBOT(whole), CoBOT(on) kept acceptable accuracy as described in Experiment 1. First, the reason that CoBOT(on) had more false negatives was due to under-approximation in our heuristics. For example, in the code "$int\ v;\ f(\&v);$" where f's definition is not available due to the incomplete application code, the actual parameter v in the caller $f(\&v)$ was considered to be initialized, because the formal parameter of f was of the reference type, and was normally initialized in the callee function f. However, the variable v might not be initialized in function f, causing false negatives. Second, CoBOT(on) had more false positives than CoBOT(whole), because CoBOT(on) considered some calls to library functions, which have already been over-written by the application code. Thus, those functions have become user-defined functions.

4 RELATED WORK

Saturn [17], Coverity [3] and Klocwork [4] are C/C++ static bug detection tools, but they do not support analysis for incomplete code. Averroes [18] is a tool that can construct call graphs for incomplete Java rather than C/C++ programs. FindBugs [19] and PMD [20] are tools supporting defect detection for incomplete Java programs by matching syntax rules. Checkmarx [21] is a commercial tool that can analyze incomplete C/C++ code. CoBOT can complement CheckMarx in both accuracy and efficiency to analyze real-world large programs. Atzenhofer and Plösch propose an approach for adding missing libraries in Java projects [22].

5 CONCLUSIONS

We present CoBOT, a static bug detection tool for C/C++. Unlike most static analysis tools, CoBOT supports bug detection without whole-program information. To effectively detect bugs in incomplete code, we propose a series of heuristic strategies in parsing, analysis and detection phases. The experiments demonstrate the effectiveness and efficiency of our tool. Our demonstration video is at: https://youtu.be/bhjJp3e7LPM.

REFERENCES

[1] CQC China homepage. Retrieved Feb 5, 2018 from http://www.sqa-bj.com/, 2018.
[2] CoBOT China homepage. Retrieved Feb 5, 2018 from http://www.cobot.net.cn/, 2018.
[3] Coverity homepage. Retrieved Feb 5, 2018 from http://www.coverity.com/, 2018.
[4] Klocwork homepage. Retrieved Feb 5, 2018 from http://www.klocwork.com/, 2018.
[5] Clang homepage. Retrieved Feb 5, 2018 from http://clang.llvm.org/, 2018.
[6] Brittany Johnson et al. Why don't software developers use static analysis tools to find bugs? *ICSE '13*, pages 672–681, 2013.
[7] CDT homepage. Retrieved Feb 5, 2018 from http://www.eclipse.org/cdt/, 2018.
[8] Sen Ma et al. Practical null pointer dereference detection via value-dependence analysis. *ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015*, pages 70–77, 2015.
[9] Yulei Sui et al. SVF: interprocedural static value-flow analysis in LLVM. *CC '16*, pages 265–266, 2016.
[10] Xiaokang Fan et al. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. *SIGSOFT '17*, pages 329–340, 2017.
[11] Ron Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
[12] Yulei Sui et al. On-demand strong update analysis via value-flow refinement. *FSE '16*, pages 460–473, 2016.
[13] Barbara G. Ryder et al. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.
[14] Thomas W. Reps et al. Precise interprocedural dataflow analysis via graph reachability. *POPL '95*, pages 49–61, 1995.
[15] Keith D. Cooper et al. Interprocedural side-effect analysis in linear time. *PLDI '88*, pages 57–66, 1988.
[16] SMTInterpol homepage. Retrieved Feb 5, 2018 from http://ultimate.informatik.uni-freiburg.de/smtinterpol/, 2018.
[17] Yichen Xie et al. Saturn: A sat-based tool for bug detection. In *CAV'05*, pages 139–143. Springer, 2005.
[18] Karim Ali et al. Averroes: Whole-program analysis without the whole program. *ECOOP '13*, pages 378–400, 2013.
[19] Findbugs homepage. Retrieved Feb 5, 2018 from http://findbugs.sourceforge.net/, 2018.
[20] Pmd homepage. Retrieved Feb 5, 2018 from https://pmd.github.io/, 2018.
[21] Checkmarx homepage. Retrieved Feb 5, 2018 from https://www.checkmarx.com/, 2018.
[22] Thomas Atzenhofer et al. Automatically adding missing libraries to java projects to foster better results from static analysis. In *SCAM'17*, pages 141–146, 2017.