Mining and Detecting Bugs in Introductory Programs

Wenchu Xu[1], Yanran Ma[2]

[1]Nanjing University, 163 Xianlin Road, Nanjing, 210093, China
[2]Nanjing Foreign Language School, 30 East Beijing Road, Nanjing, 210008, China

Abstract

For students who start learning a programming language, bugs in the introductory program hinder the learning progress. Current introductory program judge systems merely provide a pass or fail by the percentage of passed test cases, while students may be unaware of potential defects hiding in their programs. It is helpful to provide a bug detection tool for them. However, state-of-the-art bug detection methods emphasize on precision and scalability, yet developing detection methods for numerous categories of defects is typically costly. In this paper, we first conducted an empirical study to mine common bug patterns, and found that (1) most bugs in such programs are simple; (2) bug patterns rely on specific programming tasks and thus be various. According to our findings, developing precise analysis methods for each bug pattern is unrealistic and unnecessary. Therefore, this paper further proposes a static detection framework to discover software bugs in order to ease the development of bug detection while maintaining a fair level of precision. Our framework is extensible by defining bug specification for bug patterns in a specific input format. The method is then applied to real-world introductory programs and successfully detect all bugs with a false positive rate of 25.8%.

Keywords

Empirical Study, Introductory Programming, Static Bug Detection, Bug Pattern, Bug Specification

1. Introduction

Introductory programming received increasing attention both in industry and academia. Such programs, developed by students when taking introductory programming courses, are only validated by pre-defined test cases. However, passing all test cases does not guarantee that programs are free from bugs, leading to students mistakenly assume their programs are bug-free. And students may introduce similar bugs (*e.g.*, integer overflow) in real-world programs, which may lead to serious consequences [1]. Therefore, detecting bugs in such programs can help students to better understand programming and avoid similar bugs in future real-world programs.

Earlier works predominately considered program analysis for large-scale programs with complex bug patterns [2, 1, 3, 4, 5]. Carrybound [5], based on taint analysis, realizes the checking of array index out of bounds defects through backward data flow analysis. At the same time, it provides array boundary checking conditions to improve precision. Pinpoint [4] tracks the precision and scalability dilemma by symbolic expression graph, which memorizes non-local data dependence relations and path conditions. At the bug detection step, only bug-related code is precisely analyzed. This design achieves high precision and almost linear time growth. Both tools are capable of analyzing one million lines of code while keeping the analysis accurate in analyzing complex bugs.

Proceedings of 4th Software Engineering Education Workshop (SEED 2021) co-located with APSEC 2021, 06-Dec, 2021, Taipei, Taiwan
929849889@qq.com (W. Xu); ma.yanran@outlook.com (Y. Ma)

© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

In this paper, we focus on static detection methods, in order to achieve a sound result and act as a complement of program judgment systems. Unlike existing software bugs or security vulnerabilities, bugs in introductory programs may have various patterns. To better understand bug patterns in such programs, this paper first conducts an empirical study of bug patterns that introductory programs commonly have. The results demonstrate that (1) most bugs in such programs are simple; (2) bug patterns rely on specific programming tasks and thus be various.

These results of empirical study show that existing static analysis methods, albeit conceptually appealing, are not applicable for detecting bugs in introductory programs. Students without any programming background may introduce various kinds of bugs, which only occur in a certain category of programs. This raises a new challenge that, notwithstanding, the cause of each bug may be simple and trivial, detecting all of them is a non-trivial task. To the best of our knowledge, the reason is that existing static analysis methods require costly human effort to implement a sound and precise bug detector for each bug, so it is impractical to detect all trivial bugs.

In this paper, we present a simple but effective framework that is capable of detecting bugs in introductory programs. Given a bug specification, this framework first identifies potential bug locations. Then, it analyzes instructions before or after the bug location in control flow graphs. Finally, it reports a bug if instructions flow from a bug location do not meet specified constraints.

In summary, this paper makes the following contributions:

- An empirical study of student programming bugs whose goal is to find common bug patterns in

CEUR-WS.org/Vol-3062/Paper02_SEED.pdf

introductory programs.

- A static detection framework that is extensible to detect student programming bugs.
- An implementation, DBI, which we use to bugs in existing bug datasets. Results show that (1) DBI is fast, taking on average less than 2 seconds to analyze all programs; (2) DBI is effective, giving on average less than 25.8% false-positive rate in real-world programs.

The remainder of this article is presented as the following. Section 2 presents the detailed results of our empirical study. Section 3 details our analysis framework. Section 4 present our evaluation of DBI. Section 5 presents related works. Finally, section 6 gives the conclusion.

2. Mining Bugs in Introductory programs

In this section, we show the setup and result of our empirical study about common bugs in introductory programs.

2.1. Learning from practice

Source code written by novice programmers is not free from bugs even though it passes the testing of the online judge system. Such hidden bugs may make them assume that their programs are bug-free. Therefore, in order to find such hidden bugs and design an extensible bug detection framework for these bugs, we conduct an empirical study to mine common bug patterns in existing bug datasets. Such bug patterns help us to understand how they are introduced, and how to detect them.

To survey the bug patterns, we manually inspect two kinds of programs in CodeForce[1]: the first kind passes test cases and the second one does not.

2.2. Results of the empirical study

Table 1 depicts the results of our empirical study. The first column is the category of bugs. The second column is the pattern of each bug. The third column shows an example, followed by how to fix the bug. The last column shows how to fix such kinds of bugs.

According to Table 1, we find that all bugs cannot be detected in compile-time and their occurrence heavily relies on test cases. However, the test cases, provided by online judge systems, cannot cover all possible paths, resulting in missing potential bugs.

Therefore, these results demonstrate that detecting bugs is an urgent task. However, existing static analysis methods usually focus on scalability and precision.

[1] https://codeforces.com/problemset/status

```
1  void secretFunction() {
2    printf("Danger zone!\n");
3  }
4  void echo() {
5    char buffer[20];
6    printf("Enter some text:\n");
7    scanf("%s", buffer);
8    printf("You entered: %s\n", buffer);
9  }
10 int main() {
11   echo();
12   return 0;
13 }
```

Figure 1: A program with buffer overflow.

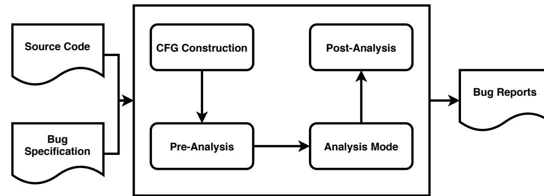

Figure 2: The framework of our method.

Specifically, their methods are proposed to analyze large-scale programs in an acceptable time budget while keeping high precision including flow-, context- and field-sensitive. Such methods are not suitable for detecting bugs in introductory programs because they require experts to develop a new bug checker. To tackle this challenge, we present an extensible framework that can detect a new bug pattern with a little human cost.

3. Detecting Bugs in Introductory programs

In this section, we first give a motivating example to show a buggy program. Then, we present an overview of DBI, followed by a detailed description of each component.

3.1. Motivating example

The code in Figure 1 looks quite safe for novice programmers. But in fact, we can call the `secretFunction` by just feeding the input with a dedicated string whose length is larger than 20. Because the vulnerable program has a buffer overflow bug introduced by `scanf`. Detecting such bugs and other kinds of bugs is a human-cost task, thus we propose our method – DBI.

3.2. Framework Overview

Figure 2 depicts the overall framework of our proposed method. It takes input as source code and bug specification, and outputs bug reports. The method consists

Table 1
Common bug patterns in the dataset.

Category	Bug patterns	Example	Bugfix
Memory related bugs	Incorrect use of `memset`	`int a[16]; memset(a, 1, sizeof(a));` `//a[0] is not 1.`	Use other API
	Buffer overflow	`int a[4]; strcpy(a, "hello world");` `//Buffer overflow`	Check the length of source and dest string

	Missing checks against external input	`scanf("%d", &nLen);`
		`for (int i = 0; i < nLen; i++)`
		`  aVal[i] = octs[i]; //Buffer overflow`
			Add checks for external input

	Use after free	`struct node *ptr = head;`
		`for (; ptr != NULL; ptr = ptr->next)`
		`  free(ptr);`
		`//ptr is freed when accessing ptr->next.`
			Add checks for external input

String related bugs	Missing '\0'	`char str[5] = {’h’,’e’,’l’,’l’,’o’};`
		`printf("str=%s\n",str);`
		`printf("strlen(str)%d\n",strlen(str));`
		`// Incorrect output`
			Add '\0'

	Missing checks after modification	`while (n[0]==’0’ && n.size()>1)` `  n=n.substr(1);`	Check string length after modification.
Float related bugs	Different precision of floating point data types	`float a=0.1+0.2;` `if(a==0.1+0.2) return true; return false;`	Replace the condition with `abs(a-b) < epsilon`

	Numerical calculations	`double sap[100] = 3.14e10; int n = 100;`
		`double sum, sqresum ;`
		`for ( int i; i < n; i ++) {`
		`  sum += sap [i ];`
		`  sqresum += sap [i] * sap [i ];`
		`} return ( sqresum - sum*sum/n) / n;`
		`// Should be zero but is` 9.761004×10^{18}
			Rewrite code

Integer related bugs	Integer overflow	`if(a+b<=c||a+c<=b||c+b<=a)`
		`  return true; // Incorrect result when`
		`a, b, c are all INT_MAX`
			Check whether math operations may cause overflow

	Conversion between Unsigned and signed	`unsigned int a = 0; int b = -1;`
		`if((a+b)>=0) return true;`
		`return false;`
		`// Should return fale but it returns true.`
			Rewrite code

	Divided by zero	`scanf("%d", &x); x = x - 5;` `int y = 100/x; // x may be zero.`	Add checks for division and modulo
Pointer related bugs	Null pointer dereference	`int *p = NULL; int q = *p;` `// Dereferencing p leads to crash.`	Rewrite code
Misuse modulo operator	Modulo negative number	`int a[10] = 0, b = -5; int c = a[b%10];` `// Array index should be larger than 0.`	Change to (b+10)%10.
	Compare modulo results	`if(100>5); if(100%10 < 5%10);` `// The comparison result is incorrect.`	Rewrite code

of 4 main components: *CFG Construction*, *Pre-Analysis*, *Analysis Mode*, and *Post-Analysis*.

CFG Construction produces inter-procedure CFGs for programs. The reaming three components, acting as the main procedure, perform the analysis against a bug specification. Its intuition is straightforward: we first search instructions that satisfy a pre-defined bug pattern. Then for each of these instructions (denoted by I_{pre}), we analyze instructions before or after I_{pre} according to the pre-defined analysis mode. Finally, a bug is reported if there is an instruction (denoted by I_{post}) along the paths from I_{pre} to I_{post} that fails to meet a pre-defined constraint.

3.3. Inputs

The inputs of the framework are the source code of a program and bug specification. The bug specification is a 4-tuple: $Spec := <BP, D, M, C>$, where BP denotes a bug pattern, D denotes directions (forward or backward), M indicates whether all paths (*i.e.*, forall) or only one path (*i.e.*, exists) should satisfy C, and C is a constraint whose violation will lead to a bug report. The four elements are used for *Pre-Analysis* (BP), *Analysis Mode* (D and M), and *Post-Analysis* (C), respectively.

BP and C are written according to the syntax of a programming language in first order logic [6]. Specifically, BP and C are defined by first order logic. Variables in BP and C are from the syntax grammar of a programming language (*e.g.*, C++).We describe these variables by the kind of statements and expressions (*e.g.*., IfStmt, ForStmt, BinaryOperator, *etc.*). For example, $I_{post} = IfStmt$ means I_{post} is an if statement.

3.4. CFG Construction

The first component in the framework is *CFG Construction*, which is a preprocessing step that generates control

flow graphs (CFG) for the input source code. Specifically, it parses the source code, constructs CFG for each function, and connects each CFG to an inter-procedure CFG (ICFG). Worth noting that we do not perform alias analysis in this step because our method naturally traces all possible aliases when analyzing bugs in each path.

3.5. Pre-Analysis

The component identifies a set of buggy instructions defined by BP. We analyze each instruction in the ICFG to determine whether the instruction meets the bug pattern. If so, the instruction is denoted as I_{pre} and the variable that saves the buggy content is denoted as Var.

For example, in order to detect the bug in Figure 1, we design $BP := I = FunCall \wedge I.Callee = scanf$. It means that we will find all instructions with `scanf` because input from users is regarded as unsafe. Therefore, a function call `scanf("%s", buffer);` is I_{pre} and its Var is `buffer` because the `buffer` saves the content of unsafe input.

3.6. Analysis Mode

We design two kinds of analysis modes: *forward/backward* (*i.e.*, D) and *forall/exists* (*i.e.*, M).

Forward/Backward. D indicates the direction of our analysis: *forward* or *backward*. *Forward* (*resp.*, *Backward*) means analyzing the instructions after (*resp.*, before) the bug pattern. The reason is to analyze two kinds of bugs: the first kind occurs when condition C is before the bug pattern while the second one is after the bug pattern. For example, given a user input, we check whether the input is valid and discard invalid input. Such checks, if exist, only be valid after the bug pattern (*i.e.*, a variable provided by users).

Forall/Exists. M indicates whether all paths or at least one path should satisfy C. The intuition of M is because one situation only requires that at least one path (denoted as $path$) that satisfies condition C to avoid introducing the bugs. In contrast, the other situation requires that all paths should satisfy condition C to avoiding introducing the bugs. Therefore, we design *forall* and *exists* modes to support the two situations.

For example, to detect the bug in Figure 1, we should check that all paths after the `scanf` should check whether the input string is valid. So D is *forward* and M is *forall*

3.7. Post-Analysis

The component checks whether variable Var in each instruction that flows from I_{pre} satisfies constraint C. I_{pre} is the buggy instruction from the *Pre-Analysis*. For each I_{pre}, we analyze instructions after (or before) I_{pre} along the ICFG. For each of these instructions (denoted as I_{post}), we construct a path (denoted as $path$) from I_{pre} to I_{post}, then check whether I_{post} satisfies condition C. If it does and the analysis mode M is *exists*, then we finish the analysis. If the analysis mode M is *forall*, then we continue analyzing other paths until all paths satisfy condition C or some path fails to meet C. If the condition C is satisfied in at least one path (*resp.*, all paths) in *exists* (*resp.*, *forall*), we will not report the bug. Otherwise, we regard I_{pre} as a buggy instruction.

For the example in Figure 1, we will not report the instruction if each path that starts from the I_{pre} (*i.e.*, `scanf` at line 7) has corresponding I_{post} that satisfies $C := I_{post} \in IfStmt.Cond$. In other words, we will not report the bug if Var (*i.e.*, `buffer`) in I_{post} is checked in a condition (*e.g.*, a `if` branch), regardless of whether the condition is correct or not.

3.8. Outputs

The output of our method is bug reports, each of which is denoted by a 4-tuple: $BR := <Loc, Var, Type, path>$. Loc is the line of I_{pre}, Var is the variable cause the bug in I_{pre}, $Type$ specifies the bug pattern of the bug (*e.g.*, buffer overflow) and $path$ gives a path from Loc to the line of I_{post}.

4. Implementation and Evaluation

4.1. Implementation

Our method is implemented by Clang[2], in order to use its basic analysis infrastructure such as parsing ASTs, constructing CFGs, analyzing instructions and analyzing paths.

4.2. Implemented bug specifications

We design bug specifications and apply it to existing introductory programs. As shown in Table 2, we implement seven bug specifications for bugs in Table 1. We explain the first two specifications in detail:

Check string length after modification. The specification is $Spec := <BP, D, M, C>$ where $BP := n[i]$ (i is an integer variable), D is set to *backward*, M is *forall*, and $C := n.size() > 1$. An example of the buggy statement is `while (n[0]=='0' && n.size()>1) n=n.substr(1);` . The correct one should swap the two predicates, resulting in `while (n.size()>1 && n[0]=='0') n=n.substr(1);`.

Modulo negative number. As for detecting Modulo operation with negative numbers, the bug specification is $Spec := <BP>$ where $BP := a\%p$ without C, M,

[2] https://clang.llvm.org/

Table 2
Bug specifications.

Bug patterns	BP	D	M	C
Check string length after modification	$n[i]$	backward	forall	$I_{post} = IfStmt$ $\wedge strlen(n) > 1 \in I_{post}.Condition$
Modulo negative number	$a\%p$	NA	NA	NA
Missing checks against external input	$I = FunCall \wedge I.Callee$ $\in \{scanf(p), get(p)\}$	forward	forall	$I_{post} = IfStmt$ $\wedge p \in I_{post}.Condition$
Divided by zero	a/p	forward	forall	$I_{post} = IfStmt$ $\wedge p\,! = 0 \in I_{post}.Condition$
Integer overflow	$v_1 + v_2 \wedge v_1 * v_2 \wedge$ $v_1 + + \wedge i \in \{v_1, v_2\}$	forward	forall	$I_{post} = IfStmt$ $\wedge i < INT_MAX \in I_{post}.Condition$
Null pointer dereference	$*p$	forward	forall	$I_{post} = IfStmt$ $\wedge p\,! = NULL \in I_{post}.Condition$
Use after free	$*p$	forward	forall	$I_{post} = FunCallStmt$ $\wedge FunCall\,! = free(p)$

Table 3
Evaluation results of synthetic programs. CSLAM is Check string length after modification, and MNN is Modulo negative number.

Subjects	LoC	CSLAM		MNN		Time (seconds)
		# of reports	# of bugs	# of reports	# of bugs	
Digits	41	2	1	1	1	0.1
Checksum	35	2	1	1	1	0.1
Smallest	29	2	2	2	1	0.1
Grade	31	1	1	1	1	0.1

Table 4
Evaluation results of real-world programs.

Subjects	LoC	# of reports	# of bugs	Time (seconds)
Buffer_Overflow	1395	14	10	6
Divide_By_Zero_Test	249	14	9	2
Integer_Overflow_Case	51	14	10	0.1
Null_Point_Case	87	8	7	0.1
Use_After_Free	152	12	10	1

and D. $(a + b)\%p$ is deemed to satisfy the BP, but $(a\%p + p)\%p$ is deemed as a bug-free statement because $a\%p + p$ must be positive.

4.3. Subjects

We evaluated DBI against synthetic and real-world programs. First, we apply it to existing introductory programs in IntroClass [7]. We randomly select four programs and inject bugs into each of them. To evaluate the real-world programs, we use programs from GitHub[3], these programs consist of known bugs to help us to evaluate the performance of DBI in a more general way.

4.4. The results of synthetic programs

Table 3 depicts the results of our method in analyzing 4 introductory programs. In all 4 programs, our method, which achieves an average 25% false-positive rate, is precise in detecting bugs. We found that one reason of false positives for the pattern of check string length after modification is caused by incomplete bug specification (*e.g.*, `length = strlen(n); length>1;` is semantically equivalent but not equal to `strlen(n)>1;`). For example, the buggy instruction is `length = strlen(n); length>1;` instead of `strlen(n)>1;`, thus no instruction that meets constraint $C := strlen(n) > 1$. The reason of false positive for Modulo negative number is also caused by imprecise BP. The instruction in the false-positive program matching the BP is $dx = num\%10$; , but the data type of num is unsigned int. To get rid of this, we need to refine bug specification to meet more situations.

[3]https://github.com/piggy10086/Nasac_benchmark

4.5. The results of real-world programs

Table 4 depicts the results of our method in analyzing real-world programs. In all programs, our tool runs relatively fast, and thousands of lines of code can be analyzed in a few seconds. The total false-positive rate is 25.8%, which is relatively low. And most of false-positive situations are caused by imprecise BP or C.

5. Related work

5.1. Static analysis

We found two strands of static methods, the first one analyzes programs with theoretical guarantee [5, 4] and the second uses deep learning models to predict bugs [2].

Carrybound [5], a static analysis tool based on taint analysis, realizes the checking of array index out of bounds bugs through backward data flow analysis and provides array boundary checking conditions to be added. Value flow analysis is way of precise static analysis, but it is not efficient enough for checking large-scale programs. To tackle this problem, Pinpoint [4] first builds fast and precise local data dependence, then it creates a new type of SVFG, *i.e.*, symbolic expression graph, which memorizes

the non-local data dependence relations and path conditions. At the bug detection step, only relevant parts were further analyzed for high precision. GINN [2] predicts bugs by learning semantics program embeddings. GINN generalizes from a curated graph representation obtained through an abstraction method. It focuses exclusively on intervals. Also, it operates on a hierarchy of intervals for scaling the learning to large graphs.

5.2. Dynamic analysis

BovInspector [1] automatically validates static buffer overflow warnings and provides repair suggestions by warning reachability analysis and guided symbolic execution. SDRacer [3], following the similar spirit of BovInspector, detects race conditions in interrupt-driven embedded software by pipelining static analysis, symbolic execution and dynamic validation. AddressSanitizer [8] is a new memory access checker that uses an efficient way to encode and map shadow memory. In addition to detecting heap space bugs, the tool can find out-of-bounds memory access bugs in the stack, global objects.

5.3. Program analysis for introductory programs

For java introductory programs, Nghi et al.. developed a static analysis framework ELP (Learning to Program) based on good programming practice experience [9]. The tool implements software metric analysis and structural similarity analysis, including cyclomatic complexity, redundant logic expression, etc., to improve programming quality and give feedback. The Investigating [10] was conducted on nearly 10 million static analysis errors in student java programs collected by Web-CAT. The results show that formatting and documentation errors are the most frequent errors. However, the two kinds of bugs, albeit frequently occur, will not lead to serious consequences. In contrast, DBI focuses on bugs hidden in programs.

6. Conclusion

This paper first conducts an empirical study showing common bug patterns written by novice programmers. However, it also finds that such bug patterns are usually simple and easy to detect but the bug patterns may rely on specific programming tasks. Therefore, to easily develop a new bug checker while keeping a reasonable precision, this paper also presents a bug detection framework for introductory programs. This method reports bugs by checking whether variables that flow from a bug pattern do not meet an expected constraint. In addition, we implement a prototype tool and conduct a case study to demonstrate the effectiveness of our approach. In future we want to improve our method to be applicable to more elaborate scenarios such as recursion and apply some more precise alias analysis algorithms to our tool.

References

[1] F. Gao, L. Wang, X. Li, Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities, in: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, pp. 786–791.

[2] Y. Wang, K. Wang, F. Gao, L. Wang, Learning semantic program embeddings with graph interval neural network, Proceedings of the ACM on Programming Languages 4 (2020) 1–27.

[3] Y. Wang, L. Wang, T. Yu, J. Zhao, X. Li, Automatic detection and validation of race conditions in interrupt-driven embedded software, in: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017, pp. 113–124.

[4] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, C. Zhang, Pinpoint: Fast and precise sparse value flow analysis for million lines of code, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2018, pp. 693–706.

[5] F. Gao, T. Chen, Y. Wang, L. Situ, L. Wang, X. Li, Carraybound: Static array bounds checking in c programs based on taint analysis, in: Proceedings of the 8th Asia-Pacific Symposium on Internetware, 2016, pp. 81–90.

[6] R. M. Smullyan, First-order logic, Courier Corporation, 1995.

[7] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, W. Weimer, The manybugs and introclass benchmarks for automated repair of c programs, IEEE Transactions on Software Engineering 41 (2015) 1236–1256.

[8] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, Addresssanitizer: A fast address sanity checker, in: 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), 2012, pp. 309–318.

[9] N. Truong, P. Roe, P. Bancroft, Static analysis of students' java programs, in: Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30, Citeseer, 2004, pp. 317–325.

[10] S. H. Edwards, N. Kandru, M. B. Rajagopal, Investigating static analysis errors in student java programs, in: Proceedings of the 2017 ACM Conference on International Computing Education Research, 2017, pp. 65–73.