An Undergraduate Course on Software Bug Detection Tools and Techniques

Eric Larson
Seattle University
901 12th Avenue
Seattle, WA 98122-1090

elarson@seattleu.edu

ABSTRACT

The importance of software bug detection tools is high with the constant threat of malicious activity. Companies are increasingly relying on software bug detection tools to catch exploitable bugs before the program is released. This paper describes a course on software bug detection techniques that is aimed at undergraduates. Courses in software verification are often taught at the graduate level and too theoretical and research oriented for undergraduates. A key component of the course is the programming assignments where students gain practical experience in creating their own software bug detection tools using a source to source converter for a subset of C++.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer Science Education* D.2.5 [**Software Engineering**]: Testing and Debugging – *symbolic execution, testing tools.*

General Terms

Reliability, Security, Verification.

Keywords

software testing, software verification, computer security, compilers, software engineering, computer science education

1. INTRODUCTION

Bugs in software can have devastating effects in today's world. Computer viruses and malicious users can exploit software bugs to run harmful code or gain access to restricted data. In addition to the threat of attacks, substantial time and money is spent keeping software up to date. One report [4] estimates that software bugs cost the United State economy $59.5 billion annually.

Software companies invest significant resources in software testing and verification in both tools and manpower. Many computer science graduates begin their careers as software testers and part of their job is spent developing or maintaining software bug testing tools.

This paper proposes a course on software bug detection tools and techniques. It describes an overview of the course as well as an infrastructure for programming assignments where students develop their own bug detection tools. The course was taught at Seattle University during the winter quarter of 2005.

After taking this course, students will have a better understanding on why testing alone is inadequate, why software bug detection tools are necessary, and gain experience writing their own tools. As a result, students can be effective members on a software testing team.

While producing software testers is not the primary goal of a computer science degree, the course provides several other benefits. Students will become better programmers as they will be more cognizant of the issues that make software verification difficult and can program in a manner that is less prone to bugs. Software developers often work with software bug detection tools and may need to annotate their source code to aid the tool.

Software verification is rich in computer science theory and the course provides several applications of theoretical topics. While the course does not attempt to overwhelm the students with theory, several different algorithms are introduced, many (such as Boolean satisfiability) of which have uses outside of software bug detection. Static bug detection is a good example of an NP-complete problem and students are able to explore how each technique presented in class mitigates this problem and still finds useful bugs. Model checking is a good example of how finite state automata are used in modeling programs and brings some practicality to a topic that students often dismiss as too theoretical.

This paper makes two major contributions:

- A description of an undergraduate course on software bug detection. While several courses in this area have been taught on this subject at the graduate level, I am not aware of a course geared toward undergraduates. Graduate courses in software verification tend to be more theoretical and research-oriented. The proposed course blends the underlying theory with practical experience and examples while exposing students to the open-ended research problems associated with software bug detection.
- A set of assignments that could be used in a course to create software bug detection tools. The infrastructure uses a source

to source converter for a selected subset of C++ and could be used for assignments in different courses.

2. COURSE OVERVIEW

The course on software bug detection was taught at Seattle University in the Winter 2005 quarter as a senior elective course. It met for ten weeks with five contact hours per week. The goal of the course is to acquaint students with techniques for detecting bugs in software with special emphasis on creating tools. A course on algorithms was the only prerequisite to the course; having prerequisites in compilers and automata theory may help as time was spent on getting the students up to speed in these areas.

This course is not to be confused with a course on software testing. While testing is a commonly used method for finding bugs in software, only one lecture was spent on the topic. The course does not go into topics such as writing tests, creating a test plan, or testing metrics. For example, when discussing dynamic bug detection – it is mentioned that the effectiveness of such a tool is dependent on how well the test suite exercises the code. Very little time is discussed on how to create such a test suite. A separate course on software testing would complement this course. It is possible to use some of the ideas in this paper to create units on software testing tools and/or software verification within a software testing course.

A textbook was not used in the course as there was no book that encompassed everything I wanted to cover in the course. Instead, a series of research papers that discussed the topics was carefully chosen. The key papers are listed as references throughout this section. Though research papers were used, the class used a traditional lecture format rather than a research seminar. However, throughout the course there were several class discussions and class participation contributed to the overall grade. One memorable discussion explored the ethical implications of malicious people using bug detection tools to find exploits in open-source software.

The course consisted of four distinct units: program analysis, dynamic bug detection, static bug detection, and assorted topics. The program analysis section focused on standard algorithms for creating a control flow graph and performing data flow analysis. This unit also illustrated that the compiler is the first bug detection tool that programmers use and explored what bugs the compiler is capable of finding. The program analysis section concluded with lectures on interprocedural analysis and alias analysis, both are necessary for high quality static bug detection and both serve as good examples of why it is difficult to obtain precise information about a program.

The second unit was on dynamic bug detection. In this unit, we focused on memory access violations, a common target for dynamic bug detection tools. The unit commenced with a lecture on testing with a focus on the different types of testing, why testing is hard [13], and where bug detection tools enter the picture. Then we explored the designs of Purify [7] and the work done by Jones and Kelly [9]. The designs are similar in that they both use additional state to track interesting behavior of a program but they differ in granularity and how they are implemented. After this section, students should have a good understanding of the tradeoffs associated with dynamic bug detection.

The static bug detection unit is the largest unit in the course and exposed students to three different approaches to detecting software faults statically: symbolic execution, constraint analysis, and model checking. Prefix [2] was used as the representative symbolic execution system and the paper discussed many of the key issues with symbolic execution: path explosion, unknown values, and pruning infeasible paths.

Constraint analysis explored the work done by Zhang and Wang [14]. While their system only deals with constraints on a single path, the paper describes an elegant algorithm for generating constraints. It also presents a solution for solving the constraints that contain both numerical constraints (such as `x < 4`) and Boolean constraints such as (`a || b && y`) using a linear programming solver and a Boolean satisfiability algorithm. While we stepped through the satisfiability algorithm, the class did not delve into the internals of the linear programming solver.

Model checking is arguably best explained using automata theory but I did not think that was the best approach for undergraduates. Instead, I gave a simplified example of how a finite state automata that represents the program could be created and explained that model checking is largely a graph reachability problem. I also described different abstraction techniques used by model checkers and why they are necessary. This section concluded with an overview of the SLAM system [1].

A unifying theme throughout the static bug detection unit explored the different ways correctness properties can be specified. Some tools, such as Prefix [2], build correctness into the system. Zhang and Wang [14] system's rely on programmer pre/post conditions and assertions. SLAM [1] uses properties that can be specified as a finite state automata using a special modeling language.

During the last two and a half weeks, assorted miscellaneous topics related to the course were covered. In one class, we discussed verifying interactive web and concurrent programs and how they are more difficult to verify than sequential programs. Another topic was the use of type systems and safe programming languages. The last topic in the course was on security and described how a buffer overflow bug can be exploited and some techniques [3] to prevent this from happening.

Many other topics are possible for this course such as software engineering approaches (coding standards, code reviews, team organizations that lead to better reliability), debugging (the art of debugging, providing enough feedback to the user when a bug is detected, debugging optimized code), and inferring bugs based on anomalies in the code [6].

For homework, the students had to complete three programming assignments (described in Section 3) while working in pairs. They also were assigned a written homework problem each class period that was due the next period. Problems varied from "manually execute the algorithm" to relatively short open-ended design problems. Exams and class participation rounded out the grading.

3. PROGRAMMING ASSIGNMENTS

The programming assignments in the course all used SUDSE, a source to source conversion infrastructure. In the first assignment, the students created analyses that were used in the later two assignments. The second assignment had students create a dynamic array checker using instrumentation. Students designed and constructed a static null dereference checker in the third assignment. While the last two assignments are both dependent on the first assignment, they are independent of each other. This section first describes the infrastructure used by the students to complete their assignments and then outlines each of the assignments in more detail.

3.1 Assignment Infrastructure

SUDSE is a source to source converter of a subset of C++ and was written using code from gcc [10] and ctool [4]. The subset of C++ was chosen to allow the students to encounter and explore the issues that make software bug detection difficult without having the coding burden of implementing the entire language. To this end, pointers and dynamic memory are included in the subset but classes, structs, templates, floating point values, and enumerated types are omitted. The only allowed types are integers, arrays (of integers or pointers), and pointers (to any of the three types).

With the exception of `cin` and `cout`, the subset could represent C. The use of `cin` and `cout` was chosen over `scanf` and `printf` because the students are taught using C++ in our computer science program and to avoid unnecessary aggravation caused by `scanf` when parsing the format string and dealing with pointer variables.

SUDSE is written in C++ using classes to represent different programming constructs. It starts by parsing in a C++ program and creates data structures that store the variables, functions, and the statements in program order. To simplify the analysis, SUDSE goes through a simplification phase that removes side effects such as increment (++) and short-circuited operators (such as &&). Many software bug detection tools and compilers go through a similar simplification phase using a simplified IR [8] or a tool such as CIL [11]. Since this course is not a compiler course, the parsing code is provided to the students and requires no modification. The data structures largely remain intact; students will need to add their own data members and functions to the various classes.

After parsing the program, SUDSE simply traverses the program and writes the content back to a file. The printing routines, which are also given to the class, serves as an example of how to traverse the abstract representation of the program.

The work that the students do is largely done between and is separate from these two phases. The one exception is a modification to the printing routines when instrumenting the program for the dynamic bug detection assignment. During the course, two hours of lecture time was devoted to the simplified C++ language and the internals of SUDSE.

SUDSE is also suitable for assignments in other courses. Clearly, it would be suitable for compiler courses, especially those that focus on classical back-end optimizations such as common subexpression elimination and loop invariant code motion. It would also be suitable for assignments in software testing. One example would be to create a statement coverage tool using an instrumentation approach similar to the dynamic array checker.

3.2 Assignment 1: Program Analysis

The first assignment had students implement standard program analyses that would be used for the latter two programming assignments.

The assignment has three parts. The first step was to identify basic blocks and create a control flow graph. In the second part, students implemented a data flow analysis for reaching definitions. In the third part of this assignment, they had to use the reaching definitions information to detect uses of uninitialized variables. This was accomplishing by creating definitions for each declaration (recall that initializers are forbidden on declaration) and marking them as uninitialized. If this definition was ever used in a statement (and not killed by an intervening statement), an error is signaled.

This assignment demonstrated that standard data flow analyses is suitable for catching bugs. One test case was constructed to show where data flow analysis breaks down. This test exposes an uninitialized use on a path through the program that is invalid and illustrates the need for distinguishing different paths of a program during testing and verification.

If the students in your course have background in back-end compiler algorithms, this assignment could be skipped and future assignments could start with a version of the tool where control and data flow is provided. Other suggestions include pointer analysis or a data flow version of the static null dereference checker.

3.3 Assignment 2: Dynamic Array Checker

The goal of the second assignment was to have students create their own dynamic array checker that validates that any array reference are within the bounds of the array. The checker also checks pointer dereferences which are equivalent to an array reference.

The design uses a pointer table that keeps track of all the pointers and arrays in the program at run time. Each table entry stores the size of the object the pointer points to and the current offset from the beginning of the object. Array variables are treated like pointers and always have an offset of zero. The table is similar to the object table described by Jones and Kelly [9]. When certain events occur in the program, the pointer table is accessed and/or updated. The pointer table can be indexed by address or by using a mangled name if you add the requirement that pointers must have names and cannot reside on the heap or within arrays.

The assignment has two distinct parts. In the first part, students modify SUDSE to add instrumentation to interesting statements. Part of the assignment was to figure out what statements needed instrumentation. Instrumentation is added using strings. If a statement needed instrumentation, the program should create a string that contains a function call with the appropriate arguments. The printing routines must be modified to actually print the instrumentation.

```
// Original code (Fig. 1A)
int bar(int x)
{
  int a[5];
  int i;
  for (i = 0; i < 5; i++) {
    a[i] = i * i;
  }
  return a[x];
}
```

```
// Simplified code (Fig. 1B)
int bar(int x)
{
  int a[5];
  int i;
  int T1, T2;
  i = 0;
  T1 = i < 5;
  while (T1) {
    a[i] = i * i;
    i = i + 1;
    T1 = i < 5;
  }
  T2 = a[x];
  return T2;
}
```

```
// Instrumented code (Fig. 1C)
#include "ptr_table.h"
int bar(int x)
{
  int a[5];
  create_array_entry((void *) a, sizeof(int)*5);
  int i;
  int T1, T2;
  i = 0;
  T1 = i < 5;
  while (T1) {
    check_array_bounds((void *) a, i, __FILE__, __LINE__);
    a[i] = i * i;
    i = i + 1;
    T1 = i < 5;
  }
  check_array_bounds((void *) a, x, __FILE__, __LINE__);
  T2 = a[x];
  delete_array_entry((void *) a);
  return T2;
}
```

```
// ptr_table.h (Fig. 1D)
void create_array_entry(void *addr, size_t size);
void check_array_bounds(
  (void *) a, int element, const char *file, int line);
void delete_array_entry(void *addr);
```

Figure 1: Example of instrumented program for assignment 2. Variables T1 and T2 are temporary variables created during the simplification process. The instrumented code in Fig. 1c includes the code of ptr_table.h, listed in Fig. 1D.

Figure 1 shows an example of an instrumented program. The original program and program after simplification are shown in Figures1A and 1B respectively. Figure 1C shows the instrumented program. Instrumentation is added to create an entry in the pointer table for the array `a`. The starting address of the table is used to access the table. Later the array is indexed twice. The first check will never result in an error but the second check will trigger an error if the value `x` passed into the function is not constrained. Finally, the array is removed from the table when the function exits because the array is no longer in scope. The file includes a header file (ptr_table.h) containing the prototypes for the instrumentation routines. A partial list of prototypes is shown in Figure 1D.

The second part of the assignment involves writing the instrumentation routines themselves. This consists of coding the functions that are listed in ptr_table.h using the pointer table design described earlier. The instrumentation routines are compiled separately to create an object file. This object file is linked when the instrumented source code is compiled to form an instrumented executable.

3.4 Assignment 3: Static Null Dereference Checker

The final assignment in the course was more open ended than the previous two assignments. Their task was to create a null dereference checker that was static. They could use any technique they liked but had to be more sophisticated than simply using data-flow analysis.

While the utopian goal of no missed bugs and no false alarms is infeasible, they had to develop a design and make intelligent decisions in order to minimize the number of missed bugs and false alarms (bug reports that are not actually bugs). Time constraints were also present - their design had to be implemented within the three weeks they had for the assignment. Students submitted a design report a week after this assignment was handed out for approval by the instructor. It forced students to think about the design up front and recognize where it was deficient. Students had to describe potential solutions to address their shortcomings. Another reason for the report was to allow me to give feedback on their design and make sure the project was adequate given the time constraints.

4. EXPERIENCES AND FEEDBACK

To gauge the effectiveness of the course, students filled out an extensive survey. The results from this survey along with my own observations form the basis of this section. Overall, the feedback was positive with virtually all of the students felt they had learned the major course objectives.

Opinions varied on the written homework assignments as some students felt that the directions were vague and unclear and there were not enough similar examples in class. The open-ended problems had some level of vagueness intentionally built into the problems. The examples in class were intended to be similar but different than the homework problems. Having a book with additional examples to fall back on would help.

The first programming assignment (program analysis) demonstrated the varied backgrounds the students had in the course. A couple of students, who likely had some compiler experience, thought the assignment was easy and not very instructional. Most everyone thought the assignment was time consuming, primarily due to the time it took to get accustomed to the source code. All of the teams, with the exception of one team that did not finish, did well on this assignment. In the future, I would likely scale this assignment back by providing the control flow graph code and having students do the data flow. I also may consider making this a pointer analysis assignment instead to give students who have compiler experience something they likely have not done before.

The second assignment (dynamic array checker) was the most successful assignment. Most students felt they learned from the assignment and the assignment was average in terms of difficulty and time. All of the teams did well on this assignment (each team received at least a B).

The third assignment had a mixed response. On the survey questions pertaining to amount learned, difficulty, and time commitment, the responses varied from average to a lot (or hard). Again, there were a subset of students that did not like the open-ended nature of the assignment and thought the directions were unclear. Most of the designs used some sort of symbolic execution. While most of the groups did what they were supposed to do, I felt that the designs and implementations could have better if they had more time. In the future, I would allow more than three weeks to complete this assignment. This can be accomplished by shortening the first assignment.

The students were also asked which topics they found the most interesting. There was no general consensus. One interesting tidbit is that students either tended to really like or really dislike the program analysis section with respect to the rest of the course. Students were also asked if there were any topics they would have liked to see but were absent from the course. A few people mentioned debugging and I agree that this a topic that should be covered in more depth as one goal of a software bug detection tool is to simplify the debugging process once a bug is detected.

5. CONCLUSION

This paper describes an undergraduate course on software bug detection tools and techniques. The course explore the underlying algorithms used by many software bug detection tools today. By using SUDSE, a source to source conversion tool for a subset of C++, students gain practical experience in developing their own software bug detection tools. Overall, the course experience was positive based on instructor observations and student feedback.

6. ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers for their insightful comments on the paper and the Seattle University students that took the course in winter 2005 for their valuable feedback.

7. REFERENCES

[1] T. Ball and S. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. Workshop on Model Checking of Software, May 2001.

[2] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. Software Practice and Experience, July 2000.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. Proceedings of the 7th USENIX Security Conference, January 1998.

[4] Ctool. http://sourceforge.net/projects/ctool/

[5] The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology report, prepared by RTI (project 7007.011), May 2002.

[6] D. Engler, D. Chen, S. Hallem, A. Chou, B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. Symposium on Operating System Principles, Oct. 2001.

[7] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. 1992 Winter USENIX Conference, Jan. 1992.

[8] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. International Workshop on Languages and Compilers for Parallel Computing, Aug. 1992.

[9] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. Proc. of the 3rd International Workshop on Automated Debugging, May 1997.

[10] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. GCC Developer's Summit, May 2003.

[11] G. Necula, S. McPeak, S. P. Rahul, W.Weimer. Cil: Intermediate Language and Tools for Analysis and Transformation of C Programs. International Conference on Compiler Construction, Apr. 2002

[12] N. Nethercote and J. Fitzhardinge. Bounds-Checking Entire Programs Without Recompiling. Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management, Jan. 2004.

[13] J. Whittaker. What is Software Testing? And Why Is It So Hard? IEEE Software, Jan/Feb 2000.

[14] J. Zhang and X. Wang. A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering and Knowledge, Volume 11, 2001.