Bug Driven Bug Finders

Chadd C. Williams
Department of Computer Science
University of Maryland
chadd@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
hollings@cs.umd.edu

Abstract

We describe a method of creating tools to find bugs in software that is driven by the analysis of previously fixed bugs. We present a study of bug databases and software repositories that characterize commonly occurring types of bugs. Based on the types of bugs that were commonly reported and fixed in the code, we determine what types of bug finding tools should be developed. We have implemented one static checker, a return value usage checker. Novel features of this checker include the use of information from the software repository to try to improve its false positive rate by identifying patterns that have resulted in previous bug fixes.

1. Introduction

Static analysis of source code to locate bugs is a well-researched area [3][9]. Static analysis has several well-known benefits. Examining the source code without actually executing the code makes the quality of test suites, a hard problem, a moot point. Static analysis also allows code to be tested that is difficult to run in all environments, such as device drivers. There are a number of systems that provide a means to write code snippets that will be used to statically check code for one type of bug or another [5][10].

It is easy for programmers to think about types of bugs that might occur, and then devise a tool to look for these bugs. However, the space of possible tools to build is very large. Instead of creating solutions and looking for bugs, we propose that efforts to build bug-finding tools should start from an analysis of the occurrence of bugs in real software, and then proceed to build tools to locate these bugs. This paper describes a study of bug databases and software repositories to determine what types of bugs static checkers should be looking for by classifying the types of bugs that are frequently reported and fixed in the code.

2. Related Work

While previous work has tried to make general predictions about faults and identify trends across the software project from software repositories, our work is concerned with specific bugs. We determine the types of bug checkers that will be useful for a code base by looking at the history of its development. We also feed data mined from the revision history back into specific bug detectors to make decisions on which flagged errors are more likely to be true errors.

There are a large number of systems in use to statically check source code for bugs. These systems have been very successful in finding various types of bugs [2]. At the very basic end of these systems are compilers that perform type checking. A step beyond these are tools like Lint that have a set of patterns to match against the code to flag common types of programming errors [13]. Systems such as metal [6] allow the user to define what type of patterns the static analysis checker should look for via state machines that are applied to the source code. Simple data flow analysis has also shown to be an effective way to statically detect bugs [10].

While static checkers are effective at finding bugs, they can produce a large number of false positives in their results. Therefore, the ordering in which the results of a static bug checker are presented may have a significant impact on its usefulness. Checkers that have their false positives scattered evenly throughout their results can frustrate users by making true errors hard to find. Previous work on better ordering of results has focused on analyzing the code that contains the flagged error [11]. Unlike previous work, we will look at data collected over the entire project and historical trends to rank our error reports.

The historical data we use are mined from revision histories stored in software repositories. Data from software repositories has been used in a number of ways to guide the software development process. The software repository data has been used to identify high-risk areas of the code based on change histories [4]. It has been claimed that data based on change history is more useful in predicting fault rates than metrics based on the code, such as length [8]. Others have worked to identify relationships between software modules by studying which pieces of the source code are modified together [7][14].

3. Mining Historical Data

The software development process produces a number of artifacts as code is written and maintained. Chief among them are bug databases and source code

Table 1: Bugs Identified in the Bug Database

NULL pointer check	3
Return Value check	4
Logic Errors/Feature Request	34
Uninitialized Variable Errors	1
Error Branch Confusion	2
External Bugs (OS or other software failed)	2
System specific pattern	1
No identified code change	153

Table 2: Bugs identified in the Software Repository

NULL pointer check	28
Return Value Check	29
Uninitialized Variable Errors	3
Failure to set value of pointer parameter	1
Feature Request	1
Error caused by if conditional short circuiting	1
Loop iterator increment error	3
System specific pattern	3

repositories containing revision histories. We use both of these artifacts to guide our research.

3.1 The Bug Database

To start our investigation, we reviewed the bug database for the Apache web server, httpd[1]. We studied the current branch of the software, which is version 2.0. We looked at the first 200 bugs that were marked as FIXED and CLOSED. We were interested in identifying the types of bugs that were fixed and matching the bug reports back to specific source code changes to classify the fixed bugs.

Our search through the bug database produced a number of interesting results. The bug reports in the bug database rarely can be tied directly back to a source code change. We were only able to tie 24% of the bug reports marked as fixed directly back to a code change. While a developer can post a comment detailing what code needed to be changed, or denote which CVS commit was created to resolve the bug, this is rarely done. Most bug reports consist of a discussion between the reporter and a developer. If the bug is fixed, the developer often ends the bug report with a short comment that contains some vague notion of where in the code the problem existed. We have also seen cases where the developer will be very specific and explain exactly what needed to be fixed or attaches a *diff*, but these seem to be the exception. Table 1 contains a breakdown of the bugs we were able to identify from the bug database.

The types of bugs we found in the bug database are worth discussing. Most of the bugs we found were logic errors or feature requests. Feature requests are just what is expected, a new feature for the software or porting a feature to a new platform. We categorize bugs where the code is correctly written to do the wrong logic as logic errors. These bugs can arise from the developer misunderstanding the specifications or not understanding how some web browsers act (in the specific case of Apache's httpd). These bugs do not lend themselves to being found via static checking. Bugs of this type are on the order of implementing the incorrect function to calculate a value, but doing the implemented calculation correctly.

All but three of the bug reports we reviewed in the bug database came from users outside the project. These reports were mostly against a released version of the software, rather than a random CVS dump of the source. Only 2 of the bug reports were marked as being reported against a CVS-HEAD version of the source code. This leads us to believe that most of the simple "statically found" bugs are taken care of by the developers before a release is made. Hence, the users exercise few of these bugs.

3.2 The Software Repository

In order to understand what types of bugs are being committed to the software repository, but not making it into a release, we inspected commit messages in the CVS repository. We looked for commit messages that contained the strings 'fix', 'bug' or 'crash' and did not have a specific bug report listed. In this way we tried to weed out as many of the bugs from the bug database as possible. Moreover, we only looked at files that had a larger number of commits to them, 50 or more. Table 2 shows the breakdown of the bugs we were able to identify from the CVS repository.

The bugs found in the CVS repository were much more amenable to identification by static analysis. While a few continued to be the result of misunderstood specifications or some other logic error, a significant number were also of the kind easily found by static analysis: a problem with the code, not with the algorithm. The two most common types of bugs found in the CVS commits were NULL pointer checks and misuse of function return values. These two types of bugs accounted for 57 of the bugs we identified in the CVS commits.

4. Static Checker

Many of the bugs found in the CVS history are good candidates for being detected by static analysis, especially the NULL pointer check and the function return value check. We chose to develop a return value checker based on the knowledge that this type of bug has been fixed many times in the past. Additionally, a return value checker can easily take advantage of data in the CVS repository to refine its results.

4.1 Return Value Checker

The return value checker we wrote checks to see if, when a function returns a value, that value is tested before being used. Using a return value can mean passing it as an argument to a function, using it as part of a calculation, dereferencing the value if it is a pointer or overwriting the value. The need for checking the return value is intuitive in C programs since the return value of a function often may be either valid data or a special error code. For example, in the case of returning a pointer the error code is often NULL. This error code could cause problems if the return value is dereferenced without being tested. If an integer value is returned, often -1 or 0 is an error code and these values should not be used in arithmetic. Even though the idea of a return value checker is not new [13], basing the return value checker on aggregate data and bug fix histories makes our approach novel.

Our checker categorizes each error it finds into one of several categories. Errors are flagged for return values that are completely ignored; the return value is never stored by the calling function. Errors are also flagged for return values that are used in some manner before being tested in a control flow statement. See Table 3 for the complete list of categories of errors our checker reports.

4.2 Ranking

The key to our checker is the ranking system used to present the output in a useful manner. Error reports are grouped by the called function. A function is ranked by how often its return value is tested before being used. This is an aggregate number generated by running the checker over all of the code in the current version of the software and tracking, for each function, the number of times the function is called and after how many of these calls the return value is used improperly. An *improper usage* of a return value is defined as either never storing the return value in the calling function or using the return value, as previously defined, before it is tested. We base our ranking on the notion that while developers produce bugs, they generally know how to use the return values of the functions they call and most often do so correctly. The more often a function has its return value checked, the more likely it is to *need* its return value checked. If a function almost always has its return value checked, the instances in which its return value is not checked are highly suspect and are good candidates for being bugs.

We also gather data automatically from the CVS commits to help with the ranking of the error reports. We search the CVS commit history to determine when a bug our checker would find has been fixed. The fact that the developer took the time to change this code suggests that it is an important change to make. We expect that the called function in such a bug fix, the function that previously did not have its return value checked, does need its return value checked before being used. Each such function we find is flagged as being involved in a bug fix in a CVS commit. We refer to these functions as being flagged with a CVS bug fix. We suspect when this function is called the return value has a valid reason to be checked before being used.

Our tool ranks errors involving functions flagged with a CVS bug fix higher than all functions not so flagged. Within each list of functions--with and without CVS bug fixes--the functions are sorted by the percent of their return values checked in the current snapshot of the software. At the top of the list then, are functions that very often have their return value checked and are flagged with a CVS bug fix.

We used a simple heuristic to determine if a CVS commit contains a return value check bug fix. The old and new versions of the committed files are both checked for return value check errors. For a given function in a file and a given function called by that function, if the new version has more return value checks that are not errors than the old version, the commit is said to fix a return value check bug for the called function. Note that simply adding an additional function call that has its return value checked makes it appear that a fix has been made.

5. Case Study

We ran a case study of our checker on the Apache httpd 2.0 source code. This is a large project with a deep CVS history. Our study was confined to the 2.0 branch and did not look into any code that resided solely in the 1.0 branch. The current snapshot contains about 200,000 lines of code and approximately 2,200 unique functions are called. These numbers include the core of the web server and optional modules. Our checker runs on Linux and we only considered modules that would run on such a system. We also included the Apache Portable Runtime (apr and apr-util) since the web server will not compile without it. The APR is a set of libraries produced by Apache to push some of the platform specific wrapper code out of httpd and give the developer a consistent set of APIs to use for common tasks.

In order to search the CVS repository for bug fixes we had to take a number of steps. For each CVS commit, we checked out the version of the code from the repository produced by that commit. We used the *configure* script supplied with the software to generate necessary files, including Makefiles. The Makefiles were used to determine the command line options needed to run the particular source file through our checker.

Table 3: Errors, CVS Bug fix flagged functions

	Checked 99% -51%	Checked 50% - 1%
Ignored (I)	22	33
Argument (A)	13	14
NULL dereference (N)	2	45
Calculation (C)	12	18
Stored, Unused (S)	8	27
Unused on Path (P)	15	9
Stored, Untested (U)	6	7

Table 4: Errors, non-CVS Bug fix flagged functions

	Checked 99% -51%	Checked 50%- 0%
Ignored (I)	67	2803
Argument (A)	48	1439
NULL dereference (N)	21	532
Calculation (C)	10	61
Stored, Unused (S)	32	429
Unused on Path (P)	17	486
Stored, Untested (U)	27	216

We successfully evaluated 5188 CVS commits to determine which functions were involved in a CVS fix to a return value check. There were 3811 more commits made to the CVS repository that we could not run through our checker. Some CVS commits would not configure correctly (1737). Some files contained C constructs that our parser could not handle, most notably having a variable number of arguments to a function (1027). The parser [12] we used was stricter with type checking than gcc. Many statements that would give warnings in gcc give errors in the parser. For instance, passing NULL, an integer, to a function that expects a void* caused the parser to raise an error. A number of commits also had true type errors where there was an actual bug checked in to the repository that resulted in a type error. The number of type errors, which caused a commit not to be checked, was 584. Also, source files raised an internal error in the parser 66 times. We were not able to track down the cause of these internal errors.

5.1 Initial Results

Our checker flagged 7,223 errors in the current snapshot of the httpd source. Each error flagged by the checker is an individual call site that has the return value produced by the called function used improperly. These 7,223 errors represent calls to 866 unique functions.

In searching the CVS commits, we found 75 functions that have a return value check bug fix and are called at least once in the current CVS snapshot. Of those 75, 41 have their return value checked 100% of the time in the current CVS snapshot (55%) and so are involved in no flagged errors. For comparison, 52% of all functions (886) had their return value checked 100% of the time. The remaining 34 functions are involved in 231 errors flagged by our checker. We consider these 231 errors likely candidates to be true errors. Note that this number of 231 does not include errors for functions with none of their return values checked, with large numbers (over 100) of unchecked return values or functions called via function pointers.

Upon inspecting these 231 errors, we believe 61 errors could be true bugs and need further inspection. The 61 bugs found in these errors gives a false positive rate of 74% for this chunk of our results (functions flagged with a CVS bug fix). See Table 3 for the breakdown of these results.

There were 86 functions not flagged with a CVS bug fix but with their return value checked more than 50% of the time in the current software snapshot. These functions account for 222 of the errors flagged by our checker. Since these functions have their return values checked more often than not, we expect these errors also to be likely candidates for being true errors. Upon inspecting these 222 errors, we believe 37 could be true bugs and need further inspection. This chunk of our results produces a false positive rate of 83%. See Table 4 for the breakdown of these results.

Overall we inspected 453 error reports and found 98 that we believe are suspicious and should be marked as a bug. This gives an overall false positive rate of 78%. The remaining 6,770 errors marked by our checker are produced by functions whose return value is checked 50% of the time or less and we expect these errors to be unlikely candidates to be true errors, thus we did not inspect them.

A false positive rate closer to 50% would be more palatable. A threshold for false positives is 50% since we would like a user to be as likely as not to find a bug when inspecting an error reported by our tool. Our technique has not yet achieved this false positive rate. However, a simple Lint-like tool would have had a higher false positive rate as each error report is given equal weight and not ranked in any way. We would have had to review each of the 7,223 errors to find the 98 bugs, which would be 73 false positives for every real bug.

5.2 A Bug Expressed

We were able to crash the httpd server by exploiting a bug found by our tool. The return value of the function ap_server_root_relative() is used directly as an argument to strcmp(). The function ap_server_root_relative() accepts two arguments, a fully qualified directory name and a filename. The return value is a char* that represents a path to a file, basically directory/filename. The return value can be NULL in a number of cases. The easiest way to get the function to return NULL is to have the fully qualified name of the file (plus NULL terminator) to be larger than 4096 bytes. In this section of the source code, 4096 appears to be the size of all the filename buffers. Obviously, if one passes a directory and filename to the function that has a combined length of more than 4096 the function will return NULL. If this happens when the return value is used directly as an argument to strcmp() httpd will crash.

6. Conclusions

In this paper we have presented a method of creating bug-finding tools that is driven by the analysis of previous bugs. We have shown that the bugs cataloged in bug databases and those found by inspecting source code change histories differ in their types and level of abstraction. Bugs listed in a bug database are generally reported by users outside of the development team and are most often reported against a public release of the software rather than a CVS snapshot. These bugs are also of a more high level nature, involved with algorithmic problems rather than simple coding problems.

We have shown that the past bug history of a software project can be used as a guide in determining what types of bugs should be expected in the current snapshot. Moreover, such data can help to recommend which of a group of bug reports are more likely to be true.

The checker we have implemented checks for function return value usage errors and uses data mined from the revision history of the software to rank the results in a useful way. With our checker we have been able to identify 98 instances in the Apache web server that we believe should be classified as bugs and need further inspection.

In the future we want to identify other static bug checkers that can benefit from information mined from a CVS repository. We also plan to refine our current static checker and run it on other software projects.

7. Acknowledgements

This work was supported in part by DOE Grants DE-FG02-93ER25176, DE-FG02-01ER25510, and DE-CFC02-01ER254489 and NSF award EIA-0080206. We would like to thank Dan Quinlan at Lawrence Livermore National Laboratory for help in using the ROSE parser.

8. References

[1] Apache Web Server, httpd. Available online at http://httpd.apache.org

[2] Ashcraft, K., Engler, D., Using programmer-written compiler extensions to catch security holes. In *Proceedings IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[3] Ball, T., Rajamani, S. K., The SLAM Project: Debugging System Software via Static Analysis, In *Proceedings of the 29th Symposium on Principles of Programming Languages* (POPL '02), Jan 2002, Portland, Oregon, USA, pages: 1 – 3.

[4] Bevan, J., Whitehead, E. J., Identification of Software Instabilities, In *Proceedings of 10th Working Conference on Reverse Engineering*, (WCRE '03) Victoria, British Columbia, Canada, Nov 13-17, 2003. pages 134-143.

[5] Engler, D., Incorporating application semantics and control into compilation, In *Proceedings USENIX Conference on Domain-Specific Languages* (DSL'97), October 15-17, 1997.

[6] Engler, D., Chelf, B., Chou, A., Hallem, S., Checking System Rules Using System Specific, Programmer-Written Compiler Extensions. *In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[7] Gall, H., Jazayeri, M., Krajewski, J., CVS Release History Data for Detecting Logical Couplings, In *Proceedings of the International Workshop on Principles of Software Evolution* (IWPSE '03), Helsinki, Finland, September 2003, pages 13-23.

[8] Graves, T. L., Karr, A. F., Marron, J. S., Siy, H., Predicting fault incidence using software change history*, IEEE Transactions on Software Engineering*, Vol 26, Issue 7, July 2000. pages: 653 – 661

[9] Heine, D. L., Lam, M. S., A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector In *Proceedings of the Conference on Programming Language Design and Implementation* (PLDI '03), June 2003.

[10] Hovemeyer, D., Pugh, W., Finding Bugs Is Easy, unpublished, http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf

[11] Kremeneck, T., Engler, D., Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations, In *Proceedings of 10th Annual International Static Analysis Symposium*, (SAS '03) San Diego, CA, USA, June 2003. pages 295-315.

[12] Quinlan, D., ROSE: A Preprocessor Generation Tool for Leveraging the Semantics of Parallel Object-Oriented Frameworks to Drive Optimizations via Source Code Transformations. In *Proceedings Eighth International Workshop on Compilers for Parallel Computers* (CPC '00), Aussois, France, Jan 4-7, 2000.

[13] *Unix Time Sharing System Programmer's Manual*, AT&T Bell Laboratories, 1979. Seventh Edition, Volume 2a.

[14] Ying, A. T. T., Murphy, G. C., Ng, R. T., Chu-Carroll, M. C., Using version information for concern inference and code-assist. Position paper for *Tool Support for Aspect-Oriented Software Development Workshop at the Conference on Object Oriented Programming, Systems Language and Applications* (OOPSLA '02), Seattle, WA, USA, November 4-8, 2002.