Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques

Chadd C. Williams and Jeffrey K. Hollingsworth, *Senior Member, IEEE*

Abstract—We describe a method to use the source code change history of a software project to drive and help to refine the search for bugs. Based on the data retrieved from the source code repository, we implement a static source code checker that searches for a commonly fixed bug and uses information automatically mined from the source code repository to refine its results. By applying our tool, we have identified a total of 178 warnings that are likely bugs in the Apache Web server source code and a total of 546 warnings that are likely bugs in Wine, an open-source implementation of the Windows API. We show that our technique is more effective than the same static analysis that does not use historical data from the source code repository.

Index Terms—Testing tools, version control, configuration control, debugging aids.

1 Introduction

Source code repositories hold a wealth of information that is not only useful for managing and building source code, but also as a detailed log of how the source code has evolved during development. If a piece of the source code is refactored, evidence of this will be in the repository. The code describing how to use the software pre and post-refactoring will exist in the repository. As bugs are fixed, the changes made to correct the problem are recorded. As new APIs are added to the source code, the proper way to use them is implicitly explained in the source code. As the code evolves and new rules develop detailing how to use internal functions, they are implicitly written into the source code, no matter if they are ever formally documented. The challenge, then, is to develop tools and techniques to automatically extract and use this information. In this paper, we focus on using data describing bug fixes mined from the source code repository to improve static analysis techniques used to find bugs.

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 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 building tools to locate these bugs. This paper describes a method where the source code change history of a software project drives, and helps to refine, the search for bugs.

The first step in the process is to identify the types of bugs that are being fixed in the software. We have done this by a manual inspection of the bug database and source code repository of the Apache Web server [29]. The next step in this process is to build a bug detector driven by these findings. The bug detector we chose to build is a function return value checker, which determines if a function's return value is tested before being used. The innovative aspect of our bug detector is that it uses information automatically mined from the source code repository to refine the rankings of the warnings it produces.

- *C.C. Williams is with the University of Maryland, 4140 AV Williams Bldg., College Park, MD 20742. E-mail: chadd@cs.umd.edu.*
- *J.K. Hollingsworth is with the University of Maryland, 4155 AV Williams Bldg., College Park, MD 20742. E-mail: hollings@cs.umd.edu.*

Manuscript received 25 Oct. 2004; revised 8 Apr. 2005; accepted 19 Apr. 2005; published online 8 June 2005.
Recommended for acceptance by A. Hassan and R. Holt.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0240-1004.

2 Preliminary Mining of Historical Data

The first step in this investigation was to review the historical data for the Apache Web server, httpd [1], to gain an understanding of what data exists and how useful it may be in the task of bug finding. For this, we did a manual inspection of the data by combing through historical information trying to determine how much information about fixed bugs exists and how easy it is to identify such information. By categorizing the types of bugs that had been fixed, we were able to drive the next stage of our work, the creation of a static analysis tool to find a particular type of bug. The details of our study can be found in [29].

Our preliminary, manual inspection of the data gave us two important insights. First, we decided that automatically mining the source code changes in the CVS repository [7], and ignoring the commit messages and bug reports, would be the most efficient way to make use of the historical information. This conclusion was based on the difficulty of correlating bug reports and the corresponding source code changes. Second, we discovered a list of bug types that were commonly fixed in the software project we studied.

3 Static Analysis Tool

Many of the bugs found in the CVS history are good candidates for being detected by static analysis, especially NULL pointer checks and function return value checks. We chose to develop a function return value checker based on the knowledge that this type of bug has been fixed many times in the past. Briefly, this checker looks for instances where the return value from a function is used in the source code before being tested. The static checker we implemented takes advantage of data that has been automatically mined from the changes stored in the source code repository to refine its results. The data that we produce from this mining is a list of functions that are involved in a potential bug fix in the software repository. The following sections describe in detail the source code checker we have implemented, how we mine the source code repository, and how the mined information is used to refine the results of the tool.

3.1 Function Return Value Checker

The return value checker determines 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 before it is tested. We also check for return values that are never stored by the calling function. Testing a return value means that some control flow decision relies on the value. The checker does a dataflow analysis on the variable holding the returned value only to the point of determining if the value is used before being tested. The checker simply identifies the original variable the returned value is stored into and determines the next use of that variable. If the variable during its next use is an operand to a comparison in a control flow decision, the return value is deemed to be tested before being used. If the variable is used in any way before being used in a control flow decision, the value is deemed to be used before being tested. In order to improve our results, a small amount of interprocedural analysis is performed. It is often the case that a return value will be immediately used as an argument in a call to a function. In these cases, the checker determines if that argument is tested before being used in the called function.

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, functions returning a pointer often return NULL as an error code. 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 may be used as an error code and if so these values should not be used in arithmetic. The idea of a function return value checker is not new [18]; however, refining the results based on data mined from a source code repository and data mined from the current version of the software (as described later) is new.

Our checker categorizes each warning it finds into one of several categories. Warnings are flagged for return values that are completely ignored or if the return value is stored but never used. Warnings are also flagged for return values that are used in a calculation before being tested in a control flow statement. Any return value passed as an argument to a function before being tested is flagged, as well as any pointer return value that is dereferenced without being tested. Table 2 shows the complete list of categories of warnings our checker reports.

While it is often the case that a function written in C returns either an error code or valid data as the return value, this is not a hard and fast rule. Some functions never return an error code and, hence, do not need their return value tested before being used. Other functions, such as printf, produce a return value that is seldom useful and nearly always ignored. These types of functions cause a static analysis tool to produce false positive warnings. Without prior knowledge, it is difficult to tell which functions do not need their return value checked. The data we mine from the source code repository and from the current version of the software is used to help determine the actual usage pattern for each function.

3.2 Mining the Source Code Repository

While we previously gathered data from the repository through a manual inspection of the CVS commit messages and source code changes, the data used by the static analysis tool is automatically mined from the source code repository by having our tool inspect every source code change in the repository. The CVS commit messages are not used when this data is gathered.

In mining the source code changes, we try to determine when a bug of the type we are concerned with is fixed. We look for a source code change that takes a function return value, which was previously not tested before being used, and adds a test of the return value. For each such bug fix, we are interested in the function called to produce the return value. We believe that such a bug fix indicates that the called function is likely to need its return value checked before being used elsewhere in the system. The fact that the programmer took the time to go back and make this change leads us to believe that it is an important change to be made.

To perform the source code mining, we use the source code checker we have developed to determine when a potential bug has been fixed by a source code change. We run our checker over both versions of the source code. If, for a particular function called in the changed file, the number of calls remains the same and the number of warnings produced by our tool decreases, the change is said to fix a likely bug. The heuristic does not try to determine if a test of a return value is removed (which may indicate the check is not needed). This may be a useful addition to the heuristic and something we plan to investigate in the future. If we determine that a check has been added to the code, we flag the function that produces the return value as being involved in a potential bug fix in a CVS commit. The end result of the mining is a list of functions that are involved in a potential bug fix in a CVS commit.

3.3 Ranking the Results

The output of the function return value checker is a list of warnings denoting instances in the code where a return value from a function is used before being tested. The user receives a full description of the warning including the source file, line number, and category of the warning. As previously mentioned, there are a number of reasons why this static analysis may produce a large number of false positive warnings. In order to make this analysis more useable, our tool tries to rank the warnings from least likely to most likely to be false positives. Two separate components are used to rank the warnings. The first is the data mined from the source code repository, the *historical context* *information*. As noted above, this is a list of functions that are involved in a potential bug fix in a CVS commit.

The second component of the ranking is data mined from the current version of the software, the *contemporary context information*. This tracks, over the entire current version of the source code, how often each function has its return value tested before being used. We determine the percentage of the invocations of a particular function where its return value is tested before being used. We use this information to gauge, from the current version of the software, how likely the programmer thought it was necessary to check the return value of a particular function.

For ranking purposes, warnings are grouped by the function called to produce the return value. The called functions, rather than the individual warnings at a call site, are ranked by our system. All warnings produced by calling a specific function are ranked together. By mining the source code repository and the current version of the software we are trying to determine the functions that are most likely to need their return values checked before being used.

The ranking is done in two parts. First, the functions are divided into two groups, those that are involved in a potential bug fix in a CVS commit and those that are not, with the former group being ranked above the latter. Next, within each group, the functions are ranked by how often their return values are tested before being used in the current version of the software. We believe that the functions most likely to need their return values checked and whose warnings are most likely to be true errors, are those that have been involved in a potential bug fix in a CVS commit and have their return values checked very often but not all the time in the current version of the software.

4 Case Studies

We have used our software repository mining techniques and static analysis tool on two different software projects. First, we looked at the Apache Web server software project. Next, we looked at the Wine project, an open-source implementation of the Windows API. Each of these projects is a multiperson, multisite effort and the resulting software is in daily use by many people.

4.1 Evaluation of Results

To evaluate our results for each case study, we produce two rankings of the warnings our static analysis tool produces. The *Naïve Ranking* contains warnings produced by calls to functions whose return value is tested before being used more than half the time in the contemporary context. This ranking is sorted by the functions' contemporary context information, and acts as the baseline for our evaluation. The ranking produced by our technique is the *HistoryAware Ranking*. The top half of the *HistoryAware Ranking* consists of *all* warnings produced by a call to a function that is involved in a potential bug fix in a CVS commit. This includes warnings produced by calls to functions whose return value is tested before being used half of the time or less in the contemporary context. This list of warnings is ranked by the functions' contemporary context information. The bottom half of the *HistoryAware Ranking* consists of all warnings in the Naïve Ranking that are not already ranked in the *HistoryAware Ranking*, ranked by the functions' contemporary context information.

For the *Naïve Ranking*, we are only inspecting functions that have their return value checked more than half of the time in the contemporary context. Since we are using this cutoff, functions that are called exactly twice and have their return value checked exactly once will never be included in the *Naïve Ranking*. We do not believe this to be a significant population. In the Apache case study, we found three functions that were called exactly twice and had the return value checked exactly once, and they all happened to have been flagged with a potential bug fix in a CVS commit. In the Wine case study, we found 11 such functions, nine of which were flagged with a potential bug fix in a CVS commit.

4.2 Apache Web Server Case Study

We ran a case study of our checker on the Apache Web server source code. This is a large project with a lengthy CVS history and we looked at nearly 3 years of CVS commits. The current snapshot contains about 200,000 lines of code and approximately 2,200 unique functions are called.[1] These counts include both the core of the Web server and optional modules. Our checker runs on Linux; thus, 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.

We successfully evaluated 6,944 CVS commits to determine which functions were involved in a function return value check bug fix in a CVS commit. There were 2,212 more commits made to the CVS repository that we could not run through our checker. The commits that would not run through our checker did so for a number of reasons. Some CVS commits would not configure correctly, for reasons discussed in Section 5.4. Some files contained C constructs that the parser we used could not handle, most notably having a variable number of arguments to a *#define* macro. Also, the parser [24] does not yet fully support parsing of GNU extensions to the C language [27]. A small number of commits also had compile errors where a file with a syntax error was checked in to the repository.

4.2.1 Special Considerations

The Apache Web server source code is unique in a number of ways. First, the code is divided into a number of pieces, many of which are optional to build. The core code is quite small (around 30,000 lines of code) and provides only the basic functionality of a Web server. All of the interesting functionality resides in modules that the user can optionally build and load at runtime. One of the challenges we faced was to ensure that, when we analyzed a source file that was part of one of these modules, we configured the source code correctly to include that module in the build process.

In addition to the optional modules, the Web server also relies on the Apache Runtime Library. The APR is a set of libraries produced by the Apache project to isolate some of the platform specific code from the Web server and give the

1. Our study was confined to the 2.0 branch.

TABLE 1
Warnings and Likely Bugs for the Apache Web Server

		Warnings	Likely Bugs	False Positive Rate
CVS bug fix flagged functions	Function checked > 50% of the time	121	38	68%
	Function checked <= 50% of the time	163	63	61%
	Subtotal	284	101	64%
Non-CVS bug fix flagged functions		283	70	75%
Total		567	171	70%

developer a consistent set of APIs to use for common tasks. This code is an Apache project outside of the Web server's source code repository and it is one of the pieces that has been most troublesome in getting source trees to configure correctly. Early in the development of the Web server, it appears, from looking at old releases of the software, that the APR was part of the Web server's repository and located in a different directory than it is today. See Section 5.6 on CVS shortcomings for a discussion as to why this is a problem.

4.2.2 Results for the Apache Web Server Case Study

Our checker flagged 6,718 function call sites in the current snapshot (taken from the CVS repository on 29 Oct 2003) of the Apache Web server source code with warnings. These 6,718 warnings represent calls to 1,779 unique functions.

In searching the CVS commits, we found 110 functions that are flagged with a likely return value check bug fix and are called at least once in the current CVS snapshot. Those functions were involved in 232 likely bug fixes identified in the source code repository. Of those 110, 58 (52 percent) have their return value checked 100 percent of the time in the current CVS snapshot and so are involved in no warnings. For comparison, 56 percent of all functions (1,001) had their return value checked 100 percent of the time. The remaining 52 corrected functions are involved in 284 warnings flagged by our checker. We consider these 284 warnings likely candidates to be true errors. These 284 warnings do not include functions whose return value is never checked, functions with large numbers (over 50) of unchecked return values, functions called via function pointers or functions whose return value is checked less than 11 percent of the time in the contemporary context. We chose 11 percent as our lower bound after inspecting warnings produced by functions down to 1 percent. We observed that all warnings below the 11 percent mark were false positives.

Upon inspecting these 284 warnings, we believe 101 warnings could be true bugs and need further inspection. By this, we mean that in the particular calling context of the warning, it was either clear the return value *could not be* safely used without being tested or not clear the return value *could be* safely used without being tested. The 101 bugs found in these warnings give a false positive rate of 64 percent for this chunk of our results (functions flagged with a CVS bug fix). See Table 1 for the breakdown of these results. The warnings produced by functions flagged with a potential bug fix in a CVS commit are broken down in Table 1 by contemporary context ranking as well.

There were 100 functions that did not have a bug fix identified by our tool in the CVS repository whose return value was checked more than 50 percent of the time in the contemporary context. These functions account for 283 of the warnings flagged by our checker. Since these functions have their return values checked more often than not, we expect these warnings are also likely candidates for being true errors. Upon inspecting these 283 warnings, we believe 70 could be true bugs and need further inspection. This subset of our results produces a false positive rate of 75 percent. See Table 1 for the breakdown of these results.

TABLE 2
Warnings Reported for the Apache Web Server

Warning Type	CVS bug fix flagged functions				Non-CVS bug fix flagged functions	
	Checked (50%-99%]		Checked (10%-50%]		Number of Warnings	Likely Bugs
	Number of Warnings	Likely Bugs	Number of Warnings	Likely Bugs		
Ignored	26	6	23	11	75	26
Argument	22	16	66	12	97	17
NULL dereference	3	3	39	26	28	10
Calculation	17	10	11	0	17	8
Stored, Unused	20	0	10	5	34	3
Unused on Path	30	1	11	7	19	1
Stored, Untested	3	2	3	2	13	5

Table 2 contains a breakdown of the warnings we inspected by category (see Section 3.1 for a description of these categories).

Overall, we inspected 567 warning reports and found 171 that we believe are suspicious and should be marked as a likely bug. This gives an overall false positive rate of 70 percent. The remaining 6,151 warnings marked by our checker are produced by functions whose return value is checked 50 percent of the time or less and we expect these warnings to be unlikely candidates to be true errors, thus we did not inspect them.

A false positive rate closer to 50 percent would be more palatable since, at this level, a user is as likely as not to find a bug when inspecting a warning 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 warning report is given equal weight and not ranked in any way. A programmer using Lint would have had to review each of the 6,718 warnings to find the 171 bugs, which would be 39 false positives for every real bug. Furthermore, the density of false positives near the top of the list is equally, perhaps more, important than the total rate. The distribution of false positives within the results is explored in Sections 4.5.3 and 4.5.4.

4.3 Wine Case Study

We conducted a second case study of our checker on the Wine source code [30]. This is another open-source project with an extensive CVS history. We mined more than 6 years worth of history. The snapshot used in the analysis was taken from the CVS repository on 14 September 2004.

We successfully evaluated 21,671 CVS commits to *.c* files to determine which functions were involved in a potential bug fix in a CVS commit. There were 18,847 more commits made to C language source files in the CVS repository that we could not run through our checker. The commits that would not run through our checked did so for a number of reasons. Some CVS commits would not configure correctly, for reasons discussed in Section 5.4. Some files contained C constructs that our parser could not handle, most notably having a large multidimensional array initialized in a declaration.

4.3.1 Special Considerations

The Wine source code presented our tools with a number of constructs that our underlying parser could not handle. These include the keyword inline, inlined assembly code, and function attributes. For the most part, we were able to write Perl scripts to patch the source code from the repository to remove these constructs. However, with over 40,000 source code revisions to C language source files, we could not inspect each one by hand to ensure that the source code would go through the parser. Certainly, at least some of the parser's failures were due to the patch scripts failing to correct the code or producing incorrect code.

The source code repository also contained a number of odd attributes. Almost all of the commits made to the repository are marked with the same author (69,654 of 70,703 are attributed in the CVS repository to the same author). This resulted in there being a much smaller number of CVS transactions identified than we expected. The Wine repository has 70,715 revisions but has only 4,971 CVS transactions (an average of 14.2 files per transaction versus 2.7 files per transaction for the Apache Web server). Since our sliding window algorithm, discussed in Section 5.3, uses the author field as a matching criterion we believe that having the same author on almost all the commits may have inflated the size of the transactions. Many of the commit messages in the repository listed an e-mail address and name that was different than the name of the author that made the CVS commit. However, this was not the rule in the repository and we did not try to mine the repository for this information. It appears that multiple people contribute to this project but a single source code librarian performs virtually all the CVS commits.

We do not expect these larger transactions to affect our results since we do not do any analysis of potential bug fixes per transaction. As long as the source trees produced by these transactions configure properly, allowing our static checker to run on the updated files, we will still be able to recover all the potential bug fixes from the CVS repository.

4.3.2 Results for the Wine Case Study

Our checker flagged 84,812 warnings in the current snapshot of the Wine source. These warnings represent calls to 11,735 unique functions. In searching the CVS commits, we found 147 functions that have are flagged with a likely return value check bug fix and are called at least once in the current CVS snapshot. Those functions were involved in 262 likely bug fixes identified in the source code repository. Of those, 50 have their return value checked 100 percent of the time in the current CVS snapshot (34 percent) and, so, are involved in no warnings. For comparison, 37 percent of all functions (4,404) had their return value checked 100 percent of the time. The remaining 97 functions are involved in 778 warnings flagged by our checker. We consider these 778 warnings likely candidates to be true errors. These 778 warnings do not include functions whose return value is never checked, functions with large numbers (over 50) of unchecked return values, functions called via function pointers or functions whose return value is checked less than 11 percent of the time in the contemporary context.

Upon inspecting these 778 warnings, we believe 260 warnings could be true bugs and need further inspection, using the same criteria outlined for the Apache Web server case study. The 260 bugs found in these warnings give a false positive rate of 67 percent for this chunk of our results (functions flagged with a CVS bug fix). See Table 3 for the breakdown of these results.

There were 513 functions not flagged with a potential bug fix in a CVS commit but with their return value checked more than 50 percent of the time in the current software snapshot. These functions account for 1,537 of the warnings flagged by our checker. Since these functions have their return values checked more often than not, we expect these warnings also to be likely candidates for being true errors. Upon inspecting these 1,537 warnings, we believe 285 could be true bugs and need further inspection. This chunk of our results produces a false positive rate of 81 percent. Overall, we inspected 2,315 warning reports and found 546 that we

TABLE 3
Warnings and Likely Bugs for Wine

		Warnings	Likely Bugs	False Positive Rate
CVS bug fix flagged functions	Function checked > 50% of the time	329	106	68%
	Function checked <= 50% of the time	449	154	66%
	Subtotal	778	260	67%
Non-CVS bug fix flagged functions		1537	285	81%
Total		2315	545	76%

TABLE 4
Warnings Reported for Wine

Warning Type	CVS bug fix flagged functions				Non-CVS bug fix flagged functions	
	Checked (50%-99%]		Checked (10%-50%]		Number of Warnings	Likely Bugs
	Number of Warnings	Likely Bugs	Number of Warnings	Likely Bugs		
Ignored	119	52	157	98	611	179
Argument	128	26	160	34	484	36
NULL dereference	31	19	21	6	103	51
Calculation	10	8	32	0	94	10
Stored, Unused	5	0	34	4	125	1
Unused on Path	21	1	40	9	53	3
Stored, Untested	15	0	23	3	67	5

believe are suspicious and should be marked as a bug. This gives an overall false positive rate of 76 percent. Table 3 shows the breakdown of these results. Table 4 contains a breakdown of the warnings we inspected by category.

4.4 Analysis of the Ranked Functions

It is informative to look at what types of functions are ranked highly by our HistoryAware metric. The first set of functions to study includes those that are ranked highest by our system. The next set of functions to study contains those functions that are checked, in the current context, 50 percent of the time or less, but are flagged with a potential bug fix in CVS. These functions produce warnings that would not be recommended for inspection except for being flagged by CVS mining.

We first look at the top functions from each of our case studies. Three of the top ranked functions for the Wine project are system supplied string manipulation functions, strrchr, strchr, and strstr. The function strrchr is also one of the top ranked functions for the Apache project. Three of the functions in the Wine results return a pointer to an already allocated data structure; they are basically lookup functions. Two of the functions in the Apache results fit this description. These functions return NULL to indicate the data was not found. An example of a bug involving one of these functions can be found in Fig. 1. If the function `ap_server_root_relative` returns the value NULL, the code snippet will cause a segmentation fault. One of the highly ranked functions in each case study is used to allocate memory, alloc_handle in Wine and malloc in Apache. The Apache results also contain two highly ranked functions that perform some complex logic and return a status code to signal if the logic failed. Another two highly ranked functions in the Apache results manipulate a data structure and return a status code to indicate success or failure.

We look next at the highest ranked functions that are checked 50 percent or less in the current context but are flagged with a potential bug fix in the CVS history. In the Apache results, three of the functions access an already allocated data structure via a pointer. Three functions perform some logic and return a status flag. One manipulates a piece of data passed to it as an argument and either returns that data or an error code. One function allocates and initializes memory. For the Wine results, five of the functions perform some type of complex logic and return a status flag. One is a system string manipulation function (strtoul). Two functions access a previously created data

```
if ((ap_server_pre_read_config->nelts
   || ap_server_post_read_config->nelts)
   && !(strcmp(fname, ap_server_root_relative(p, SERVER_CONFIG_FILE)))){
```

Fig. 1. Example return value check bug.

TABLE 5
Apache Chi-Square Calculation

	Functions checked 50% of the time		Functions Flagged with a potential bug fix in CVS		Total
Likely bugs	(38+70)	108	(101-0)	101	209
False Positives	(121-38) + (283-70)	296	(284-101)	183	479
Total	404		284		688

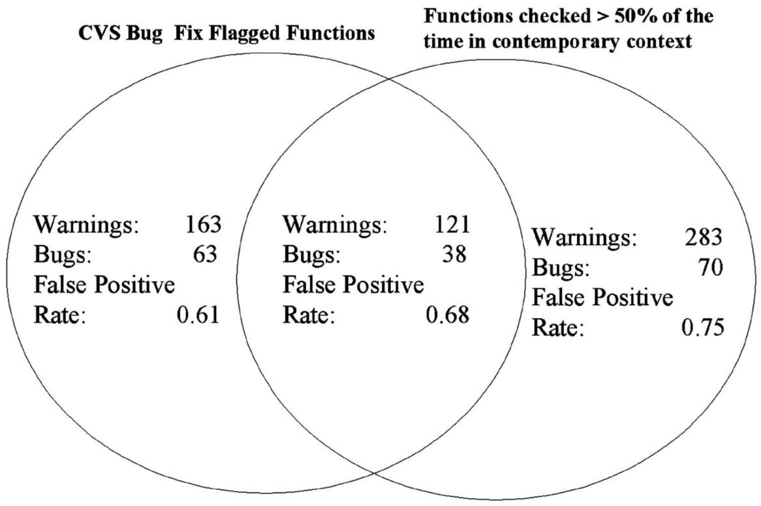

CVS Bug Fix Flagged Functions
Functions checked > 50% of the time in contemporary context
Warnings: 163
Bugs: 63
False Positive Rate: 0.61
Warnings: 121
Bugs: 38
False Positive Rate: 0.68
Warnings: 283
Bugs: 70
False Positive Rate: 0.75

Fig. 2. Division of warnings, Apache Web server.

structure. It is interesting to note that one function in each set of results is concerned with some type of locking functionality, a general mutex lock in Apache and a lock on a byte range within a file in Wine.

4.5 Effectiveness of Using Mined Data

Our goal for using data mined from the software repository is to improve the results of the static analysis tool we have built. To judge the efficacy of this approach, we need to determine whether a developer will be more likely or more quickly able to find true bugs in a list of warnings ranked with historical context information than without. To do this, we measure whether our ranking system produces a lower overall false positive rate and if the true bugs tend to cluster near the top of the list of warnings.

4.5.1 Statistical Significance

In order to evaluate our results, we ran a Chi-square test on the results of each of our case studies to determine if the improvement we see in the false positive rate due to our ranking system is statistically significant. For each case study, we compare the false positive rates of the population of warnings selected by looking at functions checked more than 50 percent of the time in the current context against the population of warnings selected by looking at the functions that are flagged with a potential bug fix in the software repository.

In performing the Chi-square test on the data from the Apache Web server case study, we determined the Chi-square value to be 6.149, which exceeds the criteria (3.84) for 95 percent confidence. The data from the Wine case study was also statistically significant. The Chi-square value was calculated to be 26.76. This also exceeded the criteria (3.84) for 95 percent confidence. In fact, the Wine data was statistically significant at the 0.001 level. Table 5 shows the number of potential bugs in each category used to calculate the Chi-square value for the Apache Web server case study. The calculations in parenthesis explain how the numbers were derived and are from Table 1.

4.5.2 Relationships of Ranking Criteria

In analyzing our results, it is important to look at where the likely bugs are found within our ranking of warnings. Our ranking is really based on two criteria. The historical context information contains functions flagged with a potential bug fix in a CVS commit. The contemporary context information measures how often a function has its return value tested before being used in the latest snapshot of the source code. For the Naïve Ranking, we have chosen to look at warnings produced by functions whose return value is checked more than half the time in the contemporary context. For the HistoryAware Ranking, we have chosen to look at all the returned warnings, except those specifically discussed before. The Ven diagram in Fig. 2 shows how the HistoryAware Ranking interacts with this notion of checking only warnings produced by functions whose return value is checked more than half the time in the contemporary context for the Apache Web server results. The left circle represents all warnings listed in the HistoryAware Ranking. The right circle represents all warnings produced by functions flagged as having their return value checked more than half the time in the contemporary context. The intersection of the two circles shows the warnings flagged by both criteria. Fig. 3 shows

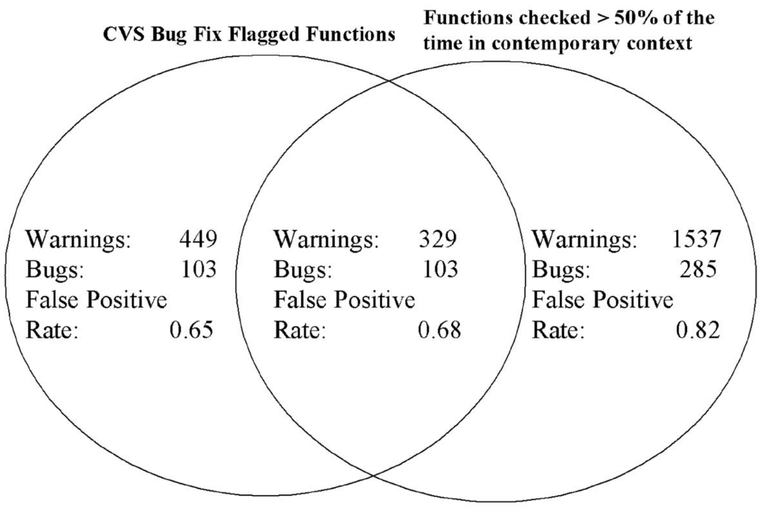

CVS Bug Fix Flagged Functions
Functions checked > 50% of the time in contemporary context
Warnings: 449
Bugs: 103
False Positive Rate: 0.65
Warnings: 329
Bugs: 103
False Positive Rate: 0.68
Warnings: 1537
Bugs: 285
False Positive Rate: 0.82

Fig. 3. Division of warnings, Wine.

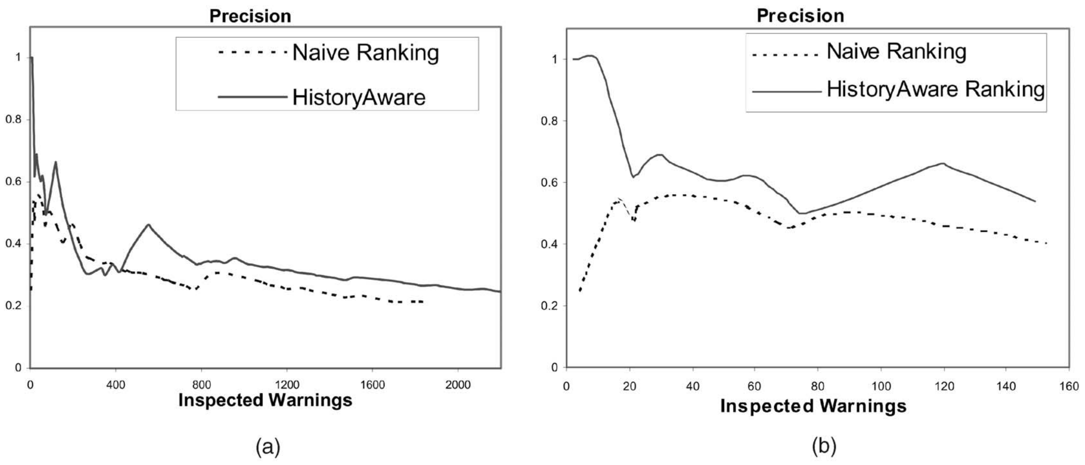


Fig. 4. (a) Precision in the Wine case study. (b) Precision in the Wine case study, detail.

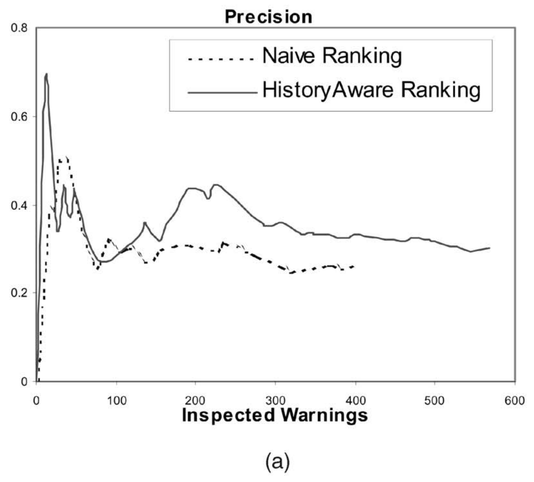
the same thing for the Wine results. Figs. 8 and 9 examine the false positive rates for the subsets of warnings produced by using cutoff rates other than 50 percent.

In both cases, the group of warnings included in the HistoryAware rankings associated with functions that do not have their return value checked more than half the time in the contemporary context have the lowest false positive rate. This seems to indicate that functions that do not often have their return value checked in the contemporary context but are involved in bug fixes in the repository produce warnings that are more likely to be real bugs. It could be that these functions do not need their return value checked in every case, but that programmers have a difficult time in understanding the cases where they do. This data invites further investigation.

4.5.3 Precision

In Figs. 4a and 5a, we plot the measure of *precision* of the list of warnings produced by running the Wine and Apache Web server source code, respectively, through our static analysis tool. Figs. 4b and 5b provide a more detailed view of the left side of the graphs, representing the first warnings inspected, from Figs. 4a and 5a, respectively. Precision measures how many of the retrieved items are relevant items. In the context of our case studies, we are measuring how many warnings we classified as likely bugs were retrieved (the relevant items) versus the number of false positive warnings (the irrelevant items). We plot precision against the number of warnings inspected. We would like to have a very high precision for the first warnings we inspect. That would indicate that the true errors are being pushed to the top of the list by our ranking system.

Figs. 4a and 5a show the precision over warnings inspected in two different lists of warnings. Fig. 4a shows that our ranking system starts out well and continues to achieve a higher precision than the list of Naïve Ranking until warning number 165. From about warning number 500 our system again begins to be more precise than the Naïve Ranking. Fig. 5a shows our ranking system starting out very precise then falling a bit below the Naïve Ranking before becoming more precise again around warning 121. In the Wine results shown in Fig. 4a, for the top 50 warnings our precision is 0.62 (meaning nearly two of every three warnings is a likely bug) while the precision of the Naïve Ranking is 0.53. In the Apache results shown in Fig. 5b, for the top 50 warnings the



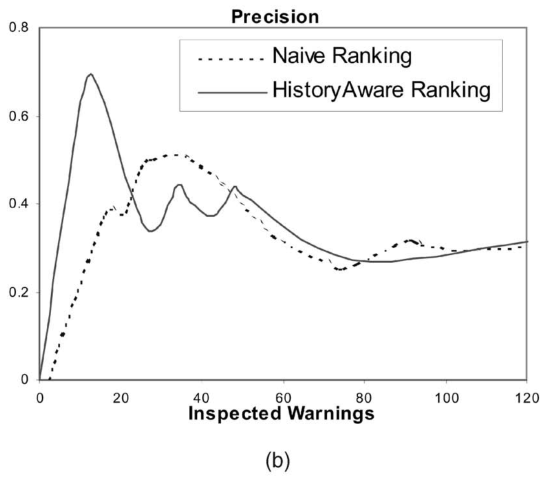


Fig. 5 (a) Precision in the Web server case study. (b) Precision in the Web server case study, detail.

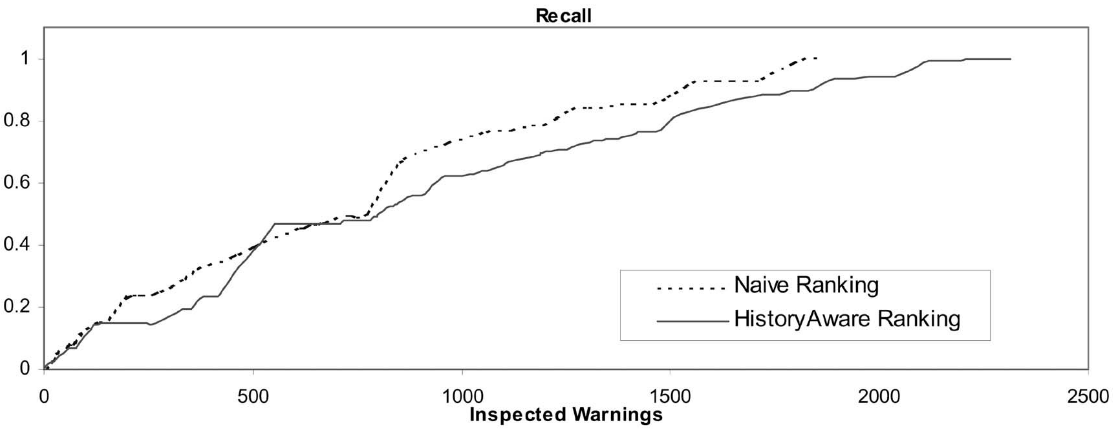


Fig. 6. Recall in the Wine case study.

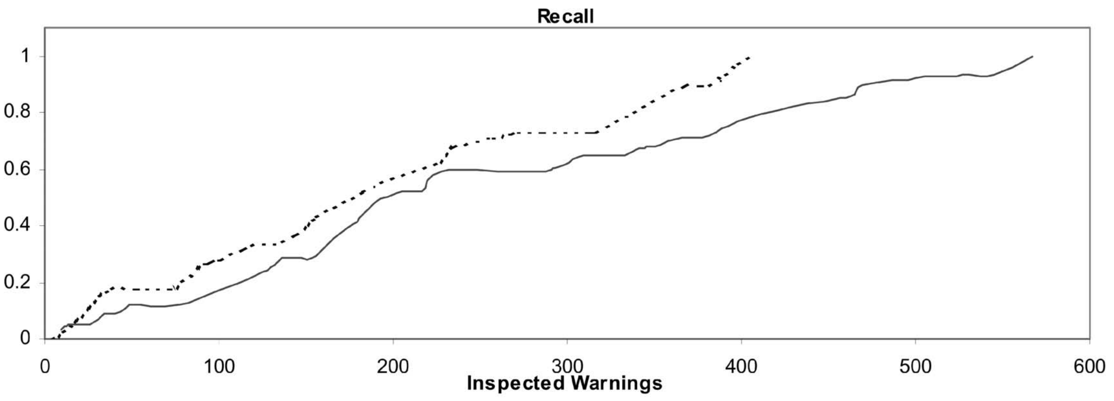


Fig. 7. Recall in the Web server case study.

precision for the Naïve Ranking is 0.32, while the precision for the HistoryAware Ranking is 0.42.

4.5.4 Recall

Figs. 6 and 7 plot the measure of *recall* of the same two sets of warnings for Wine and the Apache Web server, respectively. Recall measures how many of all of the possible relevant items have been retrieved. It is a measure of how deeply down the list of warnings a user would need to go to find some desired percentage of the relevant warnings. This is a monotonically increasing function of the number of warnings inspected. If a ranking function is effective, the plot will increase steeply near the left side of the graph. In both figures, the recall of the Naïve Ranking increases at a faster rate then the recall of the HistoryAware Ranking. This is the result of the HistoryAware Ranking, in both case studies, identifying more likely bugs than the Naïve Ranking. Since the HistoryAware Ranking has more possible relevant items than the Naïve Ranking, the recall of the Naïve Ranking increases more quickly, even though the

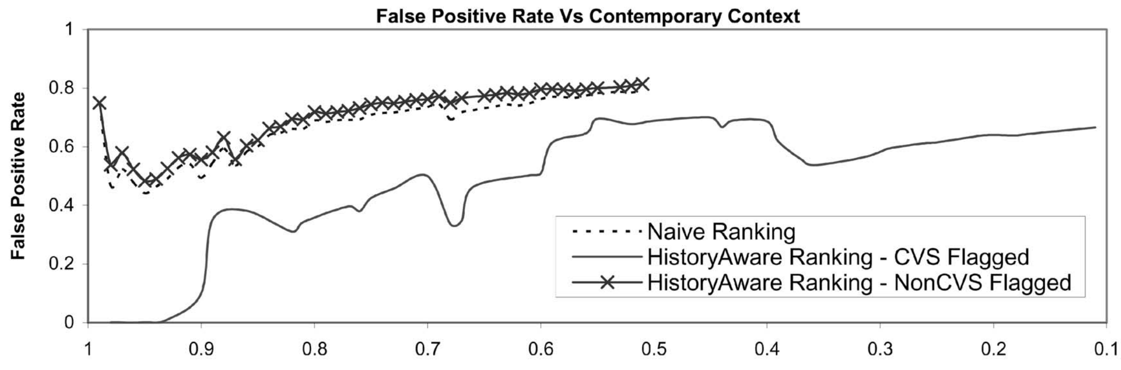


Fig. 8. False positive analysis in the Wine case study.

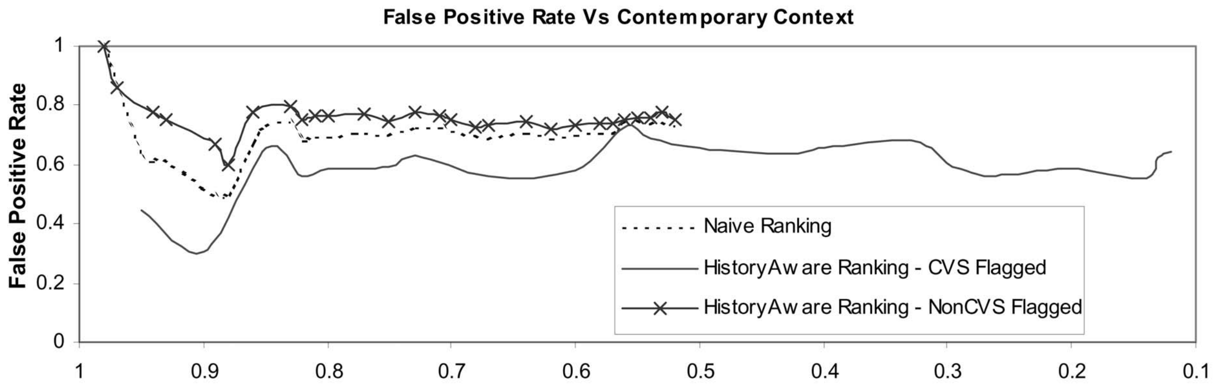


Fig. 9. False positive analysis in the Web server case study.

precision, or density of likely bugs, is higher for the HistoryAware Ranking.

4.5.5 Cumulative False Positive Rate

Figs. 8 and 9 plot the cumulative false positive rate against the measure of contemporary context for the functions that produce the warnings that are inspected. The x-axis on these two graphs is the contemporary context information—how often in the contemporary context a function has its return value checked? We were interested to see how the contemporary context information would affect the false positive rate. Thus, these graphs have three, instead of two plots on them. The *Naïve Ranking* plot is the same as was described earlier. The *HistoryAware Ranking* has been broken into two parts here, the *HistoryAware Ranking—CVS Flagged* and *HistoryAware Ranking—Non-CVS Flagged*. The CVS Flagged plot shows the results of inspecting the warnings from the HistoryAware Ranking that were produced by a function that was flagged with a potential bug fix in the source code repository. Our ranking scheme places these warnings at the top of the list. The Non-CVS Flagged series consists of the rest of the warnings produced by our ranking system, the warnings produced by functions *not* flagged with a potential bug fix in the repository and with their return values checked more than half the time in the contemporary context. These are placed at the bottom of the list produced by our ranking.

The CVS Flagged warnings have a lower false positive rate throughout the rankings in both case studies. There is also an increase in the false positive rate for the CVS Flagged series very near the 0.6 contemporary context ranking in both sets of results. There is also a decrease in the false positive rate for this series around 0.4 in the Wine results and 0.3 in the Apache Web server results. We speculate that this may indicate a class of functions whose return value needs to be checked only in particular calling contexts, thus giving these functions a ranking from the contemporary context that is in the middle of the graph. Programmers may have difficulty determining when the return values for these functions need to be checked, leading to these functions being flagged with a potential bug fix in the CVS commit.[2] However, since many of these functions only need their return values checked in particular contexts, a large number of their warnings are false positives.

While the false positive rate for our HistoryAware ranking remains high, over 50 percent, this is a considerable improvement over the original set of warnings produced by our tool and over traditional compiler warnings and tools such as Lint. This work takes a bug pattern that is nearly unusable with its number of false positives and allows it to be used with some success. More dataflow analysis and a deeper understanding of the context of function calls may help to improve the false positive rate; however, the goal of this work is to determine how much improvement historical data could provide to the results.

2. This is perhaps an indication of a maintenance issue with the code. These functions may need to be refactored to make their need of a return value check more consistent or to be better documented to inform developers of their peculiarities.

4.6 Threats to Validity

This section discusses a number of threats to the validity of our experiments. Only the first author inspected the large number of warning messages produced by our tool. The author is not an expert on either the Apache httpd source code or the Wine source code. In reviewing the warnings, an attempt was made to determine which warnings were truly false positives and which should be further inspected by an expert. The false positive rate is based on likely bugs, those bugs that we believe should be inspected by an expert. This causes our calculation of false positives to be a lower bound on the true false positive rate as determined by an expert developer. For our analysis, we have only measured the false positive rate of the warnings; we have not dealt with false negatives. In the future, we may analyze the false negatives produced by our static checker by seeding bugs in a selection of *gold standard* code to determine how many false negatives are produced. Finally, we have not tried to identify instances of function renaming. If a function *foo* had previously been named *bar*, *bar* being flagged with a potential bug fix in the CVS repository would not contribute to the ranking of warnings produced by *foo*.

5 Implementation Details

In the following sections, we give a brief description of the overall process we developed to take data from a CVS repository and produce results from our static analysis tool.

5.1 Storing Revision Histories in a Database

CVS is implemented as a layer on top of an earlier revision control system called RCS [25]. When data is stored in a CVS repository, there are two options for querying the data, operating on the RCS files directly or using CVS commands to retrieve the data. Neither of these options are particularly pleasant. Our solution was to replicate all the data from the CVS repositories we worked with in a MySQL database [28]. In addition, when we run any analysis on a source file, we store those results in the database.

Storing the data from the CVS repository in a database has been a boon to our productivity. Using the Perl DBI Interface has allowed us to easily produce scripts that interact with the database [8]. The schema we have used is a modified version of the schema described by Zimmermann [31]. The changes we made were to not store information regarding directories separate from the information stored for files and to not store CVS branch information. With regards to the directory information, we believe this more accurately represents the way CVS stores information (see Section 5.6 which describes the shortcomings of CVS). For each file in the CVS repository, the full text of each revision is stored in the database. While less efficient than storing a diff from the previous version, in this era of cheap storage devices, this makes recreating a source tree and searching the text of the revisions easier. Storing data for both case studies took less than 5 gigabytes of disk space.

A branch in CVS is a fork in the development of the source code, creating a parallel source tree that continues to be managed by the same CVS repository. A branch is often used to represent a released version of the software. This allows the software team to support bug fixes in the released software from inside CVS while still working on development tasks for the next release on the trunk. Changes made to the branch are not visible in the main trunk of the source tree unless those changes in the branch are explicitly merged back into the trunk. We do not want to analyze the same update twice: once in the branch and once as the changes on the branch are merged back into the main trunk. Therefore, we do not mine any changes made to a branch; we only mine changes made to the main trunk. Changes made to a branch are mined when changes on that branch are merged into the main trunk.

5.2 The Results Database

We store the results of running the static analysis on each version of each source file in a database. Storing the results of our static analysis in the database provides us the ability to sort, search, and inspect the results quite easily. This is particularly useful in identifying files that have failed to go through the static analysis tool or for particular time spans where updated files fail to go through the tool. The latter case pointed out an instance of a particular revision of an autoconf file in the Apache Web server source that was incompatible with the installed version of autoconf and prevented 15 transactions from configuring correctly. Fully automatic mining may eventually be possible, but currently it is an iterative process, and storing results in the database eases this process.

5.3 CVS Transactions

A series of commits may be part of a larger CVS transaction if they are all the result of one CVS command line operation. Also, some developers may issue several commit commands which are logically related (e.g., one per file or directory). Unfortunately, CVS does not explicitly store transaction information in the repository.

The goal of identifying a CVS transaction is to recreate as correctly as possible the state of the source tree that the programmer commits from when the repository is updated. We expect that the programmer's local copy of the source tree is the version that has been tested with the updated code and is a stable, working source tree. A source tree that does not encompass an entire CVS transaction may contain configuration or syntax errors as a result. These errors may prevent static analysis tools from operating properly on the source code. We have implemented a variation of the sliding window algorithm described in [31] to rebuild the CVS transactions. However, we do not consider commit messages since a programmer can provide different commit messages on a per directory basis during a single commit. Our implementation of the sliding window algorithm is based strictly on timestamps, authors, and the notion that a file cannot be updated twice by the same commit.

This approach may catch commits that are not only produced by one CVS command but by a series of CVS commands issued by the programmer in quick succession. We expect that any commits made by an author in less time than it takes to compile and build the software project are related, or at least represent a set of changes that need to be made together to maintain a usable source tree. As long as these commits are done within a reasonable amount of time the sliding window algorithm can recognize that they are part of a related transaction. Since CVS commit styles differ so much, even if CVS did record the transactions, some version of the sliding window algorithm would be required to reconstruct logical commits.

The size of the window in the sliding window algorithm determines how far apart to commits can occur and still be considered part of the same transaction. Choosing the size of the sliding window is somewhat tricky. We initially used a window of 30 seconds, thinking that the difference in successive timestamps from one CVS command could not be very large. However, as mentioned earlier, many programmers commit large numbers of files one by one or by directory, causing the logical CVS transaction to stretch beyond one CVS command and beyond what a 30 second sliding window would identify. We evaluated window sizes of 30, 180, 240, 300, and 360 seconds. We jumped from 30 to 180 seconds because the gap between the two commits that tipped us off to this problem was just over 120 seconds. The decrease in the number of transaction found was substantial between the 30-second and 180-second windows, a loss of 1,189 transactions (8 percent of the transactions found with a 30-second window). The decrease in the number of transactions was smaller for the rest of the window sizes with the differences between 180, 240, 300, and 360 seconds being 313, 212, and 198, respectively. The window of size 300 seconds seemed to provide a stable number of transactions and that was the size we use throughout this paper.

Another challenge regarding CVS transactions is to deal with overlapping transactions from different authors. Two transactions overlap if at least one file from the second transaction is updated between the first file and the last file from the first transaction. If we simply checked out the source code from the CVS repository when the last file of the first transaction is updated, we will have not only the first transaction applied to the source tree but at least part of the second transaction as well. Since our goal is to recreate the source tree seen by the programmer making the commit, the source tree that would be checked out from the repository would be incorrect. In order to produce the most accurate source tree, we need to check for overlapping transactions and be sure to rollback any commits that belong to a transaction that overlaps the transaction that we are interested in.

5.4 Interactions with the Build Environment

We need each source tree we analyze to configure correctly. Each software project we analyzed is supported on multiple platforms and passes a number of command line options to the compiler to describe which parts of the source code need to be included to compile on a particular platform. In order to do our analysis, we need to capture exactly how the Makefiles invoke the compiler for each source file we are studying. Since many of the Makefiles are generated by the configure scripts supplied in the source tree, this requires the source tree to configure properly. Further, many of the *header files* in the Apache httpd project are generated by the same configure scripts to allow for platform specific differences to be incorporated.

We wanted to avoid compiling as much of the source code as possible since the software projects we studied can take more than 20 minutes to fully build. Our solution has been to invoke *gmake* with the *-n* option, which will merely print out all the commands that the Makefile would invoke. This scheme has the added benefit of revealing which directory is the current directory while the Makefile operates. Knowing the current directory is important since the compiler may be invoked from a directory different than the one the file resides in, and the relative paths to header files in the source file may reflect that. Both our scheme and the one outlined in [10] have problems with "sneaky" Makefiles that move files generated by the make process around during the build process and do not properly specify a dependency in the Makefile between the new location of the file and the old location. We deal with this by identifying a skeleton set of code that must actually be compiled for *gmake -n* to function properly. The scheme in [10] proposes dealing with this by adding wrappers for the *cp*, *mv*, and *ln* commands. Additionally, in the case of the Apache Web server project we need to ensure, when we analyzed a file from a loadable module, the module was enabled during configuration.

Trying to configure and build the source tree raises another issue; one that we believe could be solved by a version control system. Most software projects rely on a hodge-podge of tools, libraries, and homegrown languages to configure and compile the source code. Not surprisingly, it is important to capture the history of the versions of these tools to aid in the mining of the source code of the main software project. Different tools and libraries may be used through the lifetime of the project or, more likely, the versions of the tools and libraries used will change. Unfortunately, as these support packages change the new versions may be incompatible with the previous versions. Some projects include the tools they use during the build process in their repository. However, widely available tools like autoconf and gmake are rarely archived. It would be extremely helpful if the version control system tracked not only source code produced by the project but the tools and libraries that the project relies on. Programmers could then denote clearly in the repository when support for a new compiler was established, or when the autoconf files were updated to be used with a particular version of autoconf. As it stands now, a project may have an ad hoc way of tracking which versions of which tools it relies on, or this data may be buried in the commit messages of the project. Providing a standard way to store this information, which is vital to configuring and building a source tree, seems like a natural fit for the version control system.

5.5 Computational Costs

Beyond the overhead of getting the source tree to build correctly is the computational cost of running the analyses. To mine the data from the source code repository required us to run our tool over tens of thousands of revisions of files (between the two projects we have analyzed almost 50,000 revisions) stored in over 20,000 CVS transactions. Checking one CVS transaction, from extracting the source tree to storing the results in the database, took roughly 4 minutes. Most of this time was taken by the configure scripts. Our tool takes about as long as a compiler to analyze a source file and never took more than a few tens of seconds on any of the source files we checked.

In order to mine all these CVS transactions, we used a 64 node Linux cluster managed by the Portable Batch Scheduler (PBS). We only used 20 processors at a time so as not to overburden the database machine. Dedicating a subset of the processors on a 64-node cluster to such a task may seem extravagant, but the acquisition cost of such a cluster is about the same cost of salary and overhead for a midlevel developer for a year. In a production environment, we would expect commits to be mined as they are made to the repository with the results stored away for later analysis.

5.6 Challenges in Dealing with CVS

CVS has become ubiquitous in the open-source community as the tool of choice for source code version control. As a tool for developers, it is a wonderful piece of open-source software that fills a critical need in the software development community. However, as a tool to support the mining of a large source code repository that may go back many years (or decades) it could be more helpful if it addressed a small number of key issues.

One of the tasks required to mine a CVS repository is to run our static analysis tool over a snapshot of code from the repository at a particular moment in time. It would be terrific if we could reliably checkout a source tree that would compile and build. Of course, there are times when a syntax error is committed to the repository or the source code is in a state of flux and will not build. However, there are many times when CVS itself stymies the compilation of the source tree because of the information it does not store.

First, there is no way to move a file from one directory to the next while maintaining the file's revision history. To move a file in the repository, the user is forced to edit the repository by hand or use CVS commands to remove the file from its original location and add it in the new location. This erases all mention of the file's existence somewhere else and causes problems in checking out a version of the source tree that should contain the file in its original location. Obviously, this source tree will not compile and an analysis that requires the source tree to compile will fail. Slightly less problematic is when a header file or some other widely used file is moved and the repository updated. We did see this problem in some of the early versions of the Apache Web server source code. A file or directory had been moved and as a result the source tree would not build and very few of the files would go through our static analysis tool successfully.

Related to the above problem is the fact that there is no way to rename a directory. If the user wishes to rename a directory, either every file will need to be moved to a new directory or the CVS repository will need to be edited by hand to update the name of the directory. Each of these solutions brings on a number of problems very similar to those discussed for moving a file.

Finally, as discussed above we need to reconstruct CVS transactions from individual file commits to ensure a consistent source tree.

6 Related Work

The main thrust of our work has been to investigate how we can use data mined from source code repositories to improve software development. While others have tried to make general predictions about faults and to identify trends across the software project from software repositories, our work is concerned with discerning specific properties of the code. We use these properties to refine static source code checkers when looking for specific bugs. Much of the other work in this area has dealt with historical data from within a single company or from a series of class projects, the data we have mined came from large-scale open-source projects.

Bevan and Whitehead show how static dependency graphs can be augmented with data from software repositories to identify areas in the code that need to be refactored due to code evolution [4]. Ostrand et al. describe a tool that automatically looks at the characteristics of a software project and, using historical data, predicts which files are likely to contain a larger number of faults [22]. Graves et al. use change histories to understand how code ages. They define code to be aged if its structure makes it unnecessarily difficult to understand or maintain. They posit that data based on change history is more useful in predicting fault rates than metrics based on the code, such as size [14].

Purushothaman and Perry present a study of small changes to determine the impact they have on the software [23]. Specifically, they look at the properties of the change itself, number of lines added, removed, or modified, rather than properties of the code being changed, to determine how small changes affect the code.

Chen et al. have built tools to allow the user to search for information via a grep-like command that operates on the source code and CVS commit comments [5]. This allows developers to search the source code via CVS comments for source code snippets.

Other work has focused on trying to locate common updates to source code to identify successful maintenance strategies. Rysselberghe and Demeyer document using clone detection techniques to identify frequently applied changes to the source code [26]. These changes are then studied to identify possible maintenance activities, such as refactoring. They also propose matching frequently applied changes to bug reports to help to identify bugs in the code and, possibly, solutions to these bugs. Hassan and Holt propose tracking changes of more fine grained entities, namely, function, variable, or data type, to determine how changes propagate from one entity to another [15].

Zimmermann and Weissgerber have studied the tasks necessary to extract usable data from a software repository [31]. This includes looking at the task of identifying a transaction, which is a set of commits to a collection of files made by the same developer at the same time. They also look at the problems associated with merges of branches in a CVS repository. German describes an algorithm to reconstruct transactions that is very similar in [13]. They include a file revision in a transaction if that file revision has the same author and same log message as other files in the transaction and that file revision is at most t seconds away from at least one other file revision in the transaction. The addition of that file revision also must not produce a transaction that is more than T seconds long. The main difference from the algorithm discussed in [13] and [31] is the addition of the maximum length of the transaction. Our goal in recreating a transaction is to produce a *buildable* source tree from the repository. Different CVS mining tasks may require different properties from a source tree produced by a CVS transaction. As we discuss in Section 5.3, our algorithm does not take into account the log messages for commits since, for various reasons, the log messages may differ within one transaction.

While our algorithm was driven by recreating every CVS transaction applied to the source code, other work has focused on tying bug reports or modification requests to CVS commits. Fischer et al. analyzes CVS commit messages for problem report identification numbers to determine links between the software repository and bug tracking system [12]. They use filenames found in software patches associated with problem reports as well as identification numbers to rate the confidence of each link they produce. Fischer and Gall show that their algorithm is able to link at least 50 percent of the problem reports to CVS commit messages [11]. Cubranic has done work in building project memory, tying together data that describes the source code from multiple sources [6].

Some work has focused specifically on uncovering bugs in source code by looking for violations of program specific rules [20]. Matsumura et al. describe a case study that shows 32 percent of failures detected during the maintenance phase of a software project were due to violations of implicit code rules. The implicit rules used to check the source code were generated by 'expert' programmers and had not previously been described in design documents.

Ferenc et al. have proposed a framework for capturing how the Makefiles invoke the compiler for each source file in open source software projects [10]. Their scheme, in part, creates a wrapper script that is called by the Makefiles in place of the compiler. The script receives the command line arguments the compiler would receive and records them for later use by the analysis tools as well as using them to invoke the compiler.

Static analysis of source code to locate bugs is a well-researched area [3], [16]. 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 [9], [17]. These systems have been very successful in finding various types of bugs [2]. The simplest 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 [18]. Systems such as metal 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 [9].

While static checkers are effective at finding bugs, they can produce a large number of false positives in their results. Therefore, the order 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. Those tools with few false positives at the top of their report will likely be perceived by users as more effective. Previous work on improving ordering of results has focused on analyzing the code that contains the flagged error [19].

7 Conclusions

In this paper, we have shown how data mined from a source code repository can improve static analysis tools. Furthermore, we have compared the results produced by a static analysis tool using our HistoryAware ranking historical data to the results produced using a naïve ranking technique that only looks at data from the current snapshot of the software. We demonstrated that the value added by the data mined from the software repository is statistically significant and that the precision of highly ranked items is much better than the naïve technique.

From a preliminary investigation of historical data, we have shown that the bugs cataloged in bug databases and those found by inspecting source code change histories differ in the types and level of abstraction. The users, not the developers, of the software, often report the bugs found in a bug database. This affects the type of bugs reported and in which phase of software development these bugs are found. Inspecting the software repository provide much better data. Repositories record all the bugs fixed, from every step in the development process. The knowledge gained from the preliminary investigation was used to guide the reminder of our work.

The next step of our work was to implement a static source code checker and to implement a system to automatically mine data from a source code repository. With our static checker we have been able to identify 178 likely bugs in the Apache Web server and 546 likely bugs in the Wine source code. The two case studies we present show our technique to be more effective than the same analysis without using historical data. Our technique for ranking warnings had better precision than a similar technique that was based on data gleaned only from the current snapshot of the source code. In each of our case studies, the false positive rate of the rankings produced by our technique was consistently lower then that of the naïve ranking.

In the future, we need to explore expanding this research to other types of bugs. It would be interesting to investigate software repositories to see how often bugs found by such tools as FindBugs [17] get fixed. Another matter to investigate is how to use this data to refine the results of such tools, as we have done here, or to predict where these tools would be useful, in a similar manner to previous work [21] that try to gain knowledge from bug reports and change requests. Automatically discovering where these bugs are fixed, rather than relying on bug reports, may provide a more complete set of data.

Acknowledgments

This work was supported in part by Department of Energy (DOE) Grants DE-FG02-93ER25176, DE-FG02-01ER25510, and DE-CFC02-01ER254489 and US National Science

Foundation award EIA-0080206. The authors would like to thank Dan Quinlan at Lawrence Livermore National Laboratory for help in using ROSE. They would also like to thank Mustafa Murat Tikir for his kind advice during the preparation of this paper. They also would like to thank the anonymous reviewers for their extremely thorough and thoughtful comments.

REFERENCES

[1] Apache Web Server, httpd, available online at http://httpd.apache.org, 2004.
[2] K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proc. IEEE Symp. Security and Privacy,* May 2002.
[3] T. Ball and S.K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," *Proc. 29th Symp. Principles of Programming Languages (POPL '02),* pp. 1-3, Jan. 2002.
[4] J. Bevan and E.J. Whitehead, "Identification of Software Instabilities," *Proc. 10th Working Conf. Reverse Eng. (WCRE '03),* pp. 134-143, Nov. 2003.
[5] A. Chen, E. Chou, J. Wong, A.Y. Yao, Q. Zhang, S. Zhang, and A. Michal, "CVSSearch: Searching through Source Code using CVS Comments," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM '01),* pp. 364-373, Nov. 2001.
[6] D. Cubranic, "Project History as a Group Memory: Learning from the Past," PhD thesis, Univ. of British Columbia, 2004.
[7] CVS—Concurrent Versions System, available online at http://www.cvshome.org, 2004.
[8] A. Descartes and T. Bunce, *Programming the Perl DBI.* O'Reilly, 2000.
[9] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System Specific, Programmer-Written Compiler Extensions," *Proc. Fourth Symp. Operating Systems Design and Implementation,* Oct. 2000.
[10] R. Ferenc, I. Siket, and T. Gyimothy, "Extracting Facts from Open Source Software," *Proc. 20th Int'l Conf. Software Maintenance (ICSM '04),* pp. 60-69, Sept. 2004.
[11] M. Fischer and H. Gall, "Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data," *J. Software Maintenance and Evolution: Research and Practice,* vol. 16, pp. 385-403, Nov./Dec. 2004.
[12] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and Relating Bug Report Data for Feature Tracking," *Proc. 10th Working Conf. Reverse Eng. (WCRE '03),* pp. 90-99, Nov. 2003.
[13] D.M. German, "An Empirical Study of Fine-Grained Software Modifications," *Proc. 20th Int'l Conf. Software Maintenance (ICSM '04),* pp. 316-325, Sept. 2004.
[14] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.,* vol. 26, no. 7, pp. 653-661, July 2000.
[15] A.E. Hassan and R.C. Holt, "Predicting Change Propagation in Software Systems," *Proc. 20th Int'l Conf. Software Maintenance (ICSM '04),* pp. 284-293, Sept. 2004.
[16] D.L. Heine and M.S. Lam, "A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector," *Proc. Conf. Programming Language Design and Implementation (PLDI '03),* June 2003.
[17] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *Companion of the 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04),* Oct. 2004.
[18] S. Johnson, *Unix Time Sharing System Programmer's Manual,* seventh ed. vol. 2A, AT&T Bell Laboratories 1979.
[19] T. Kremeneck and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," *Proc. 10th Ann. Int'l Static Analysis Symp. (SAS '03),* pp. 295-315, June 2003.
[20] T. Matsumura, A. Monden, and K. Matsumoto, "The Detection of Faulty Code Violating Implicit Coding Rules," *Proc. Int'l Workshop Principles of Software Evolution (IWPSE '02),* pp. 15-21, May 2002.
[21] T. Menzies, J.S. DiStefano, C. Cunanan, and R. Chapman, "Mining Repositories to Assist in Project Planning and Resource Allocation," *Proc. Int'l Workshop Mining Software Repositories (MSR '04),* May 2004.
[22] T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Where the Bugs Are," *Proc. 2004 ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA '04),* July 2004.
[23] R. Purushothaman and D.E. Perry, "Towards Understanding the Rhetoric of Small Changes," *Proc. Int'l Workshop Mining Software Repositories (MSR '04),* May 2004.
[24] D. Quinlan, "ROSE: A Preprocessor Generation Tool for Leveraging the Semantics of Parallel Object-Oriented Frameworks to Drive Optimizations via Source Code Transformations," *Proc. Eighth Int'l Workshop Compilers for Parallel Computers (CPC '00),* Jan. 2000.
[25] RCS, available online at http://www.cs.purdue.edu/homes/trinkle/RCS/index.html, 2004.
[26] F. Rysselberghe and S. Demeyer, "Mining Version Control Systems for FACs (Frequently Applied Changes)," *Proc. Int'l Workshop Mining Software Repositories (MSR '04),* May 2004.
[27] R.M. Stallman, *Using the GNU Compiler Collection.* GNU Press, 2004.
[28] M. Widenius and D. Axmark, *MySQL Reference Manual Documentation from the Source.* O'Reilly, 2002.
[29] C.C. Williams and J.K. Hollingsworth, "Bug Driven Bug Finders," *Proc. Int'l Workshop Mining Software Repositories (MSR '04),* May 2004.
[30] Wine, available online at http://www.winehq.org, 2004.
[31] T. Zimmermann and P. Weissgerber, "Preprocessing CVS Data for Fine-Grained Analysis," *Proc. Int'l Workshop Mining Software Repositories (MSR '04),* May 2004.

Chadd C. Williams received the BS degree in computer science from West Virginia University in 1998 and the MS degree in computer science from the University of Maryland in 2002. He is a graduate student in the Computer Science Department at the University of Maryland, College Park. His research interests include software evolution and program comprehension.

Jeffrey K. Hollingsworth received the BS degree in electrical engineering from the University of California at Berkeley in 1988. He received the MS and PhD degrees in computer science from the University of Wisconsin in 1990 and 1994, respectively. He is an associate professor in the Computer Science Department at the University of Maryland, College Park, and affiliated with the Department of Electrical Engineering and the University of Maryland Institute for Advanced Computer Studies. His research interests include instrumentation and measurement tools, resource aware computing, high-performance distributed computing, and programmer productivity. Dr. Hollingsworth's current projects include the dyninst runtime binary editing tool and harmony—a system for building adaptable, resource-aware programs. He is a senior member of IEEE and a member of ACM.