ABSTRACT

Title of Document:	USING HISTORICAL DATA FROM SOURCE CODE REVISION HISTORIES TO DETECT SOURCE CODE PROPERTIES
	Chadd Creighton Williams, Doctor of Philosophy, 2006
Directed By:	Professor Jeffrey K. Hollingsworth, Department of Computer Science

In this dissertation, we describe several techniques for using historical data mined from the source code revision histories of software projects to determine important properties of the source code. These properties are then used to improve the results of various bug-finding techniques as well as to provide documentation to the developer. We describe a method to mine source code revision histories, in this case CVS repositories, to extract relevant information to be fed into a static source code bug finder for use in improving the results generated by the bug finding tool. We apply this technique to the CVS repositories of two widely used open source software projects, Apache httpd and Wine. We show how source code revision history can be used to reduce false positives from a static source code checker that identifies the misuse of values returned from a function call. A method of mining source code revision histories for the purpose of learning about project specific idioms is then discussed. Specifically, we show how source code revision history can be used to

identify patterns of calling sequences that describe how functions in the software should be used in relation to each other. With this data, we are able to find bugs in the source code, document API usage and identify refactoring events. In short, this dissertation shows that it is possible to automatically determine meaningful properties of the source code from studying source code changes cataloged in the software revision history.

USING HISTORICAL DATA FROM SOURCE CODE REVISION HISTORIES TO DETECT SOURCE CODE PROPERTIES

By

Chadd Creighton Williams

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair
Professor Victor Basili
Assistant Professor Jeffrey Foster
Assistant Professor Michael Hicks
Professor Martin Loeb

Dedication

For my parents.

Acknowledgements

I first need to thank my advisor, Dr. Jeffrey K. Hollingsworth. His immense patience and support made this work possible. Much of a graduate student's success is a reflection of the advisor, and I hope my work reflects well on Dr. Hollingsworth.

My advisory committee also deserves a round of thanks. They provided invaluable advice on how to improve my dissertation. Dr. Bill Pugh, a member of my proposal committee, also needs to be included in this group. His influence through my entire research career was second only to that of my advisor's.

My friends here at Maryland and elsewhere deserve thanks for helping to keep me sane as I went through all the trials of graduate school. These people include my housemates through the years (all seven of them), my fellow Dyninst API workers and anyone who visited my house so that I could cook chili or grill meat (nothing is more therapeutic). They are too numerous to mention here.

There are a number of fellow graduate students who deserve special mention. Mustafa Tikir, Brian Postow and Byran Buck all gave me wonderful advice, based on their experiences, on how to be a successful graduate student. Without their advice I'm sure I would have repeated all of their mistakes, rather that just a few of them. I hope that I have been able to offer similar advice to the newer, younger graduate students I have met over the last few years.

Finally, my family has been extremely supportive throughout my entire educational career. While they may have been mystified as to why it took so long for me to graduate (a confusion I shared at times), my parents and sister offered nothing but encouragement and much needed optimism. For this I am eternally grateful.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Source code revision 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 new code is stored in the repository alongside the buggy code. As new APIs are added to the source code, the proper way to use them is implicitly explained in the source code.

The proper way to use an API or how to invoke a function can be viewed as rules or properties describing how to use the source code that has been written. Understanding these rules is vital for the developers to produce correct code. Having a documented set of properties also allows for the automatic analysis of the source code to determine where these properties may not be correctly used. The challenge in dealing with these rules is that they evolve over time. Changes to the source code may change the rules by adding or removing functions, changing the implementation of a function or rewriting a data structure. As the code evolves and new, system specific 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 goal of this work is to determine if it is possible to identify these rules by analyzing the change history of the source code.

System specific rules can be used in a number of ways. Using these rules to find bugs via static analysis has been very successful. However, the challenge is always to correctly document all the system specific rules. This is usually left to an expert, a senior developer on the project or a group of developers, each one familiar with a different subset of the code. This is an unsatisfactory way to derive these very important rules.

Understanding the source code, how to use an API for example, is another application of these rules. As software grows and changes new rules evolve that the source code must obey. Unfortunately, few of these rules are formally documented and often new developers to the project must discover these on their own or rely on instruction from veteran developers. However, these rules that guide the source code changes are implicitly defined in the source code. More importantly they have been added to the software; hence they show up as a modification in the source code repository. Automatically extracting these properties and rules from source code modifications could allow projects to formally document these rules as time goes by. Alternatively, modifications to the source code could be checked for violations of these rules before a change is allowed to be added to the source code repository.

The thesis of this dissertation is that these properties can be extracted from software repositories and used to benefit the developers. To validate this thesis, we introduce techniques by which we examine the changes made to the source code and use this data to refine and enhance current bug finding techniques as well as using this data to document the proper use of the source code. Each of the tools we implement uses the same technique to mine the source code for a particular property.

The tools mine each version of each file to determine where a property is present in the file. We then identify which instances of the property in the later version of the file are new instances created by the source code change. The tools we produce will track how and when new instances of these properties are created in the source code.

We first discuss the infrastructure we built to facilitate mining the source code repositories. This infrastructure deals with extracting the source code from the repository and storing it in a database, recovering valid source trees from the repository for each CVS transaction and gathering configuration information needed to parse the source files successfully. While a seemingly simple set of tasks, doing this automatically for source trees of open source projects that date back through several years is quite a challenge. We must deal with changing operating system header files, buggy support tools, configure scripts written for specific (older) versions of the operating system and changes to the structure of the source tree. The goal of this infrastructure is determine what the local source tree looked like when a developer made a commit to the repository. Being able to recreate this snapshot increases the chances that the source files will correctly run through our static analysis tools. Storing the results of mining each version of each file in the database allows many of our tools, or parts of our tools, to be built as database queries. This allows quick and easy access to the data.

We present our work in applying our techniques in two chapters of this dissertation. First, we study using the bug fix history gleaned from the source code repository to improve the results of a static bug finding tool. The bug fix history is used to rank the warnings found by the tool in an attempt to increase the number of

likely bugs found in the top of the rankings. The particular bug fix we are looking for involves the return value check bug. The bug consists of instances where a function's return value is not checked against an error code before being used in the code. In C code, it is often the case that the return value of a function is either valid data or some type of error code. In the cases where the return value could be an error code, the return value should not be used without first determining it is not the error code. Our mining tool examines the source code repository to determine where a bug of this type is fixed and which function produces the return value involved. The latest version of the code is then inspected for bugs of this type and the warnings produced are sorted using the historical information mined from the repository. To evaluate this work, the precision of this sorted list is compared to the precision of a sorting that does not use historical information to rank the warnings. For both of the software projects we studied we found that the sorting that uses historical information has a better precision and that the difference is statistically significant.

The second property that we mine from the repositories is function usage patterns. This pattern describes how functions are invoked in the source code with respect to each other. We examine the code to determine pairs of functions that are commonly called together in the code and additionally are often involved in a data flow relationship. A number of statistics are calculated that measure how the new instances of the patterns are created and how likely adding one of the function calls in the pattern is to create that pattern in the code. These patterns are used in three ways. High confidence patterns are used to search the latest version of the code for instances where the pattern is broken. To evaluate this we produce a list of warnings sorted by

historical information and compare the precision of the top of this list to the precision of a sorting that does not use historical information. In the two open source projects we studied, we found that using historical information to sort the warnings produces a better precision and that the difference is statistically significant. Mined patterns are also used to document an API by selecting out patterns that contain only functions defined in the particular API. To evaluate this we mine patterns for an internally defined API and an API for an external library. Because the internal APIs are poorly documented it is difficult to strictly evaluate the quality of the patterns defined for the API. The external APIs we have built patterns for are well documented. Finally, by examining how new instances of patterns enter the code over time we can identify refactoring events. When a refactoring is identified we determine where the refactoring was not applied correctly and our bug finding tool would have raised a warning. We also look for instances where the pattern was entered into the code in a broken form after the refactoring and later fixed.

For our evaluation, we apply our techniques to two large open source projects as well as a set of student projects produced for an introductory C class at the University of Maryland. The two open source projects we used are the Apache httpd web server and the Wine open source implementation of the Windows API. The fact that these two projects are widely used, live software projects, each with many years of CVS data, drives many of the challenges we discuss in the chapter outlining the infrastructure developed to support this research.

Contributions

The main contributions of the research presented in this dissertation are: It is possible to identify important properties that describe the source code by examining source code changes. We show that our technique of using data mined from the source code repository to augment a static analysis tool makes that tool more effective than the same static analysis that does not use historical data from the source code repository. Our results show that the improvement gained by using the historical information is statistically significant. Finally, we show that mining the source code changes is a valid way to document, for the developers, the ever changing state of the code.

Chapter 2: Related Work

The work in this dissertation consists of applying static analysis tools to each version of each file in the revision history of software projects. The first section in this chapter will discuss work done with the static analysis of code on a single version of the source code. Included in this section is a discussion on static analysis of source code for the purpose of finding bugs. The research dealing with using system specific rules to guide static analysis is particularly relevant to this work. The next section of this chapter deals with research in the area of studying the source code revision history. Here we review work that describes both the engineering challenges involved in analyzing data in the revision history as well as how this data has been put to use. The last subsection of this chapter deals with dynamic analysis of source code and the benefits and tradeoffs with respect to static analysis.

2.1 Static Analysis

One of the first tools available to statically check source code for errors was lint [42]. In addition to performing more strict type checking than most C-compilers, lint also checks for a number of other code patterns that may lead to bugs. Unused variables and functions are flagged with warning messages. Uninitialized variables are also flagged as warning. Lint has a number of other built in patterns that it uses to identify potential bugs in source code. It does not, however, have any mechanism to allow the user to specify new patterns to search for in the source code. Many of the patterns identified by lint are now flagged as a warning by compilers if strict warning options are used. One of the largest drawbacks to using a tool such as lint is that the

warnings produced by the tool are presented to the user in a non-sorted list. The user can select different levels of errors to search for, but lint has no way to suggest which of the warnings are more likely to be actual bugs.

A high level description of a static analysis of the SLAM tool is laid out by Ball and Rajamani [9]. This tool is concerned with determining if a program obeys "API usage rules." A key idea from this work is that the system specific rules are inferred from static analysis of the source code rather than being specified by an expert programmer.

ASTLOG is an early system that allowed the user to specify code patterns to identify in C and C++ code [16]. This tool provides an interface to allow the user to match particular patterns of abstract syntax tree nodes to an AST built from the source code of a program. While this is a general purpose tool for identifying particular code patterns, it has been incorporated into the PREfast tool which is used to search for bug patterns in code.

PREfix is a system developed by Bush, et al, to statically analyze source code for the purpose of finding dynamic errors [12]. This system was built with a number of goals in mind. All the information should be derived directly from the source code and not rely on user annotations. The analysis should be performed on possible paths through the source code only, requiring more in-depth analysis to prune impossible paths. Bush describes a number of case studies of the use of this tool, finding a number of errors in real live software projects. The bugs they describe are not system specific bugs but general bug patterns that can be applied to any software project.

Other systems have been developed that do allow the user to specify specific patterns that should be flagged with a warning in the source code. Systems such as metal [24] allow the user to supply patterns to match against the source code and flag as warnings. The patterns the static analyzer should look for are encoded via state machines that are then applied to the source code. This system has been used to find a large number (500) of errors in real software projects [24]. Additionally, metal has been shown to be effective in the specific domain of finding security bugs in software [6]. System specific security rules were encoded in patterns and flagged as warning in the source code. This application of the static analyzer found over 100 errors in real software projects. While most of the work with the metal tool places the burden of defining the system specific rules on the user, the system was also used to try to infer system specific patterns that should be checked by examining the source code that is being inspected for errors [24]. Searching for system specific rules consists of building up *beliefs* based pre-existing templates. A statistical analysis of how often a belief is applied correctly in the source code. As more contradictions of the belief are found in the source code the belief is scaled back or discarded as being unreliable. One of the templates used by this tool is very similar to the function usage pattern *called after* that we describe in Chapter 5.

Holzmann describes a static checking system, Uno [40], that searches for a list of predefined patterns and allows the user to add their own program specific patterns to this list. The user defined extensions in Uno are written in simple ANSI-C functions, not a special language as in metal. Also, the extensions are interpreted on the fly and do not require the entire tool to be recompiled to modify the available extensions.

This work briefly describes the application of Uno to a set of real software projects in which a number of errors are detected. The Uno tool requires the developer to define the rules to apply to the code and offers no automatic detection.

Heine and Lam describe a static analysis tool to automatically detect memory leaks in C and C++ programs [39]. They develop a type system to formalize an ownership system to determine when an allocated object is unowned and therefore leaked. Their analysis is flow- and context-sensitive. This tool looks for a very specific set of errors and does not identify system specific errors or provide a way for the developer to extend its set of warnings.

The prospect of using very simple static analyses to detect bug patterns is explored with FindBugs in [41]. FindBugs uses a large number of predefined bug patterns that are applied to the bytecode of a Java class file. The level of complexity of these bug patterns run from a simple examination of the signatures of the class files to a linear scan of the bytecode to the generation of a control flow graph to allow data flow analysis. This work has uncovered hundreds of errors in a number of production software projects.

Some work has focused specifically on uncovering bugs in source code by looking for violations of program specific rules. One such system is described by Matsumura, et al., in [49]. They describe a case study that shows 32% 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. They characterize implicit rules in a number of ways. These are rules that are not described in specification or design documents, as they have evolved over

time as the source code has changed. Implicit rules are specific to the particular software project and are not merely style guidelines. This work highlights the need to be able to automatically identify implicit rules as they develop in the source code.

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 [44]. They work off the idea that a particular bug pattern that causes a number of warnings to be produced probably is producing a large number of false positives. The assumption is that the programmer generally programs correctly and if one particular checker finds a large number of bugs the checker itself is incorrect or misunderstanding the code. This work emphasizes the notion the source code that is being checked for errors can provide information regarding which warnings produced by a tool are most likely to be true bugs.

Recent work [14] has looked at classifying properties generated by a static or dynamic analysis tool in order to rank them in order of importance. In this work, machine learning techniques are used during a training phase to teach a classifier which properties are associated with buggy code. The classifier then grades each property reported during an analysis phase to point the user to the properties that most likely reveal buggy code. This method also only looks at the current version of the source code to provide a ranking of the warnings.

Mandelin, et al., describe a method for providing the developer sample code snippets to retrieve an object of a particular type, based on a query from the developer that specifies the input and output types [48]. Their tool mines the set of APIs available to the code and a set of example programs that use these APIs rather than a program's change history.

The PR-Miner tool described in [45] mines software for implicit programming rules using a data mining technique called frequent itemset mining. The items in the set that are examined by this tool include function calls, variables and data types. These frequent item sets are then used to search for detects where the set is incomplete in the code. This tool searches only the current version of the source code to determine these item sets.

Pinzger and Gall identify patterns to recover software architecture [59]. They use code patterns specified by the user, and data describing the associatations of these patterns, to reconstruct higher-level patterns describing the software architecture. Their technique relies on a good deal of existing knowledge at the source code level to define the patterns. The strength of this tool is in finding relationships between modules within the source code.

The need for information sharing in large, geographically distributed open source software projects has been studied. Gutwin, et al., studied the need for *group awareness*, knowledge about who is doing what is the project [35]. One of the aspects of awareness they describe is *feedthrough,* which is defined as observations of changes to project artifacts to indicate who has been doing what.

2.2 Software Revision History Mining

We mine historical information from revision data stored in software repositories. Others have used the data from software repositories in a number of ways to enhance the software development process. Research has been done to apply this data to monitor the progress of ongoing software projects, identify high-risk areas of the code and identifying common changes to the source code.

Research in mining source code revision histories can be along a number of axes, depending on what part of the repository is being analyzed and at what level, and for what purpose the research is done. The first axis to consider is the how the mined data is to be used. This axis has three points: using the historical data as a test bed, trying to learn about the structure of a change and trying to learn about the source code changes. Using the repository data as a test bed allows researchers to apply a technique to various revisions of the source code, make predictions and then examine how well those predictions are borne out. Predictions that can be tested by reviewing the historical data may include where bugs will occur, which warnings produced by a bug finding tool will be fixed, or where code will be refactored. These predictions may or may not be made based on historical information mined from the repository. Identifying the structure of the change may be used to match developers to a piece of code, track how a development team uses the repository, e.g. what prompts a commit, or identifying areas in the code that require the most and least number of changes. Examining historical data to learn about the source code on a deeper level can be used to elicit specific maintenance strategies, identify common types of refactorings, or determine how system specific properties evolve in the code.

The next axis pertains to what is examined for each change. At the highest level, the statistics of the commit can be studied. This includes investigating what files were changed, the commit message(s) associated with a change, the author of the change or the size of the change. This may be done to study the interaction of developers with the source code and each other, identify sets of files commonly changed together, or determine how the size of a change affects program correctness.

The next step down involves looking at what parts of the source code change in each commit. This may involve looking at what functions contain changes, what new functions are created or looking all the way down to what variables and function calls have been touched. This may be done to study sets of source code entities that need to be changed together, tracking how entities are added and removed from the code or determining which developer is responsible for each entity.

Finally, at the lowest level, researchers want to see how the different aspects of the source code have changed. This involves defining source code level properties and determining how they are involved with the code change. This may involve looking for a specific type of bug fix, a call pattern for a set of functions or how source code changes redefine how modules interact within the code. The deepest analysis of the source code occurs with research in this area.

The work described in this dissertation falls into the category of evaluating the source code after each change to identify how the code has changed for the purpose of understanding system specific properties. This is a much deeper analysis of each change than much of the previous work in the area of revision history mining. While much of the other work has tried to make general predictions about faults and 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. The data mined from the revision history is fed back into specific bug detectors to make decisions on which flagged errors are more likely to be true errors. The data is also used to document the source code. The properties discovered are presented back to the developers to highlight how these properties have changed and provide a starting point for understanding the source code.

A number of papers have studied how to use data from software repositories to monitor the progress of an ongoing software project for management purposes. Menzies, et al., describe a method for studying software repositories and defect logs to determine good defect detectors [50]. Their defect detectors "input structural metrics of code and output a prediction of how faulty a code module might be" [51]. They mine historical data including static code measures of software modules and over eight years of defect logs for these modules to produce the defect detectors. They then apply these defect detectors to other projects to show that they remain useful across software projects. Menzies, et al., use the revision history as a way to validate their defect detectors, not to learn anything about the source code in the repository. This is a common use of software repositories: applying a tool or technique against every revision of the code and validating the output against how the source code changes in subsequent versions.

Purushothaman and Perry present a study of small changes to determine the impact they have on the software [60]. They look at change histories and defect logs to

determine how small changes affect the code. 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. A significant difference between Purushothaman's and Perry's work and ours is that we look into the code to see how it is changing rather than just focusing on the characteristics of the change.

Other researchers have used data mined from software repositories to identify high-risk areas of the code based on change histories. Bevan and Whitehead show how static dependence graphs can be augmented with data from software repositories to identify areas in the code that need to be refactored, as a result of changes over time to the software [10]. Bevan and Whitehead, by building dependence graphs, show that analyzing not just that a change occurred, but how the source code changed gives useful data to answer a specific question (where refactorings should occur).

Hassan and Holt propose a system that dynamically produces a list of the top ten subsystems most likely to have a fault [37]. The heuristics used to produce the top ten list are based on metrics measured from the software repository. The metrics are completely based on characterizing the change rather than analyzing how the source code changes. They then compare the lists produced to faults found in the software over a four year period in the same manner described by Menzies, et al., [50].

Ostrand and Weyuker propose 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 [56]. This is a continuation of their earlier work in [57], where they outlined the model used to make their predictions. Again, this work only looks at the characteristics of the change as opposed to the actual source code

change. 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 make the claim that data based on change history is more useful in predicting fault rates than metrics based on the code, such as length [34]. This work also only evaluates the characteristics of the change to make assessments of the state of the source code.

Others have worked to identify relationships between software modules by studying which pieces of the source code are modified together. Ball, et al, show how studying the statistics of the software repository can be used to gain an understanding of the structure of the source code [8]. They used metrics specific to the repository, such as which files are modified together, to gain an understanding of the structure of the source code. Gall, et al., present a study in which they analyzed the software repository for the purpose of determining, at a class level, relationships between code [31]. The revision history is mined to determine when classes were added to the software and how they changed over time. This paper looks at the source code changes at a semantic level, rather than just tracking changes to files. Zimmerman, et al, present in a tool which, based on revision histories, suggests to developers which other files need to be modified, based on a change that developer makes [76]. This tool predicts files, functions and variables that need to be changed along with the pending change. This requires a deeper analysis of the source code changes than just looking at change statistics recovered from the repository. Ying, et al., present work to identify associations between software artifacts based on their change history [74]. They outline a tool based on this knowledge that would recommend to the developer

artifacts that may need changed during modification tasks. This work combines an analysis of the statistics of the change as well as a deeper analysis of how the source code changed.

Still others have built tools to allow the user to search for information via grep-like commands that operate on the source code and CVS commit comments [15]. This allows developers to search the source code via CVS comments for source code snippets [19].

Other work has focused on trying to identify common updates to source code in the software repository for the purpose of recovering successful maintenance strategies. Rysselberghe and Demeyer [63] document using clone detection techniques to identify frequently applied changes to the source code. These changes are then studied to identify 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 [38]. Both of these analyses take place at the source code, rather than the change level.

Zimmerman and Weissgerber have studied the tasks necessary to extract usable data from a software repository [75]. This work lays down a foundation of how to meet the engineering problems associated with extracting and analyzing source code from the repository. The problem is framed specifically in terms of extracting source code from a CVS repository [18]. This includes looking at the task of identifying a transaction, which is a set of commits to a number of files made by the same

developer at the same time. This data is not stored by CVS and must be reconstructed when the data is mined. They also look at the problems associated with merges of branches in a CVS repository.

Work has also been done to mine both bug tracking systems and version control systems to aggregate and store this data, allowing developers to more easily see how the data stored in bug database entries and source code version control system relate to each other [28]. This allows the developer to perform simple queries against the aggregated data to study the evolution of the source code. The system described by Ohira, et al., [54] also aggregates data from many sources, including bug databases, mailing lists and source code revision histories to provide up-to-date metrics describing the state of the source code. The data is visualized as a number of charts and graphs that allow project managers to keep track of the progress of the software development. This is motivated by the difficulty in relating specific bug reports to source code changes and vice versa. Providing a unified interface and a connection between the bug database and the source code changes gives the developers a good deal more help when they need to understand how the code has changed.

Godfrey, et al., have described the types of studies that can be made on a source code revision history [33]. They identify four broad approaches to the study of software evolution and study the goals and methodology for each. The four types of studies they outline are: coarsely-grained longitudinal case studies of growth and evolution, finely-grained case studies of origin analysis, case studies of code cloning and tracking how build architectures and software manufacturing-related artifacts change over time.

Much work with software revision histories has been done on open source projects or proprietary software [64]; there have also been studies on student projects to try to determine how students develop code [46]. Each type of software project brings a different set of challenges. Student data is small and easy to work with but can be unreliable. Proprietary software may be well documented and well controlled, but not everyone has access to such software. Open source projects are available and present good case studies for large, geographically distributed projects, but are often poorly documented and unwieldy to build.

Livshits and Zimmermann examine method calls that are added in a CVS commit to a file [47]. They build function pairs based on these added function calls, even if the function calls are added to different methods. Their work has been on Java source code and takes into account that two methods on the same object are likely related. After function call patterns are discovered, they are validated during a dynamic analysis phase. In contrast, our work focuses on the function usage patterns that are created between existing function calls and added function calls, revealed by analyzing the control flow graph, by a change to a particular function. We also use a data flow analysis component to identify meaningful patterns rather than dynamic analysis.

Xie and Pei have done work to mine frequent API usages from open source repositories and suggest code snippets based on a query from the developer [73]. This query consists of method name, class name or package name. The output of their tool is a set of API usages in terms of function call sequences. The latest version of the source code in the open source repositories are mined for API usages, but

source code changes are not used in this process. This work is similar to [48], with the addition of mining many software project to collect API usages of external APIs.

Kim, et al., describe a method by which Java classes are classified by their structure [43]. These classifications, called micro patterns, are based on the type of data and how it is exchanged between the class and outside objects. They examine various open source projects to determine how often the classification of a class changes between source code changes and which classifications of classes are likely to have software defects.

2.3 Dynamic Analysis

Dynamic analysis, the analysis of data produced by executing a program, has also been widely used in the context of program understanding and bug finding. Specifically, a number of dynamic analysis tools have been built to elicit or verify system specific properties of the source code, including call sequences and data invariants. The following works describe various uses of dynamic analysis in program understanding and bug finding.

Ball describes the concept of dynamic analysis of a running program in [7]. The paper describes two analyses: frequency spectrum analysis and coverage concept analysis. Frequency spectrum analysis counts the number of executions of each path through each function during a run of the program. Counting the frequency of execution of various parts of the source code allow the developer to compare and contrast these separate parts. The author describes a case study in which the parts are compared by high versus low frequency, similar frequencies and specific frequencies. The first two allow for the comparison of the code to itself to discover parts that are

related or not. The last allows the developer to look for specific patterns in the execution if some program specific knowledge is available. Taken together, these analyses can help the developer to understand and decompose a program. Coverage concept analysis attempts to produce "dynamic control flow invariants" for a set of executions. Comparing these to statically derived invariants can guide the development of or changes to a test suite to afford a higher quality test environment.

The Daikon tool [26] dynamically discovers data invariants from data gathered during the execution of a program. Daikon instruments a program to trace the value of variables during an execution of the program. The data produced is then tested for a set of invariants. Any invariant that holds true with sufficient support is then reported to the user. The authors demonstrate that the invariants discovered by their tool can be used in a number of ways to test software, validate software changes and check the adequacy of the test suite. In a later work, Nimmer and Ernst use Daikon to generate invariants that are then turned into ESC [53] annotations and placed in the source code. The ESC tool is then run on the source code to statically detect violations of the generated invariants.

The tool, DIDUCE, described in [36] discovers and checks program invariants on the fly and reports violations of these invariants to the user. The goal of this tool is to detect errors in the software and to identify the cause of these errors. The on the fly nature of DIDUCE, and its narrower focus, separates it from Daikon. During execution, DIDUCE gathers the value of variables and checks to ensure they adhere to previously discovered invariants. When execution begins, DIDUCE starts with a set of very strict invariants. These are relaxed as data is gathered that invalidates the

invariants. The user is also notified to the existence of a violation of an invariant when one needs to be relaxed. A confidence metric based on how many times it has been successfully evaluated to help filter out spurious relaxations of invariants until a correct set is derived. During the early phases of execution, many invariants are relaxed until a near steady state is achieved. When an invariant with high confidence is relaxed, this can point to a bug in the software, as the authors show with a few case studies presented in the paper.

Ammons, et al, describe a method of discovering specifications of a program, with respect to its interactions with an API or abstract data type, by monitoring its runtime interaction with the chosen abstraction [3]. The system extracts rules from watching the interaction of the program with the abstraction and codifies these rules in a state machine. Later executions of the program can then be verified against the generated state machines to check for bugs. The authors present a small case study that shows their method was able to correctly generate specifications, e.g. the generated specification matched some rule already documented by the developers. Also they were able to find a few bugs in the software they studied by analyzing program executions that did not verify properly.

Whaley, et al, propose a system that uses finite state machines to model the interface to a class [67]. They describe a system that includes static analysis to discover illegal call sequences, dynamic analysis to extract models from executions of programs and a dynamic model checker to verify the code adheres to the generated models. The main focus in this work is “large object-oriented component-level software” rather than low-level C code.

Chapter 3: Infrastructure

The analyses we outline in this dissertation act on a source file and use a C parser to build an abstract syntax tree. This abstract syntax tree is the representation of the program on which the analysis is actually performed. In order to apply our tool, we need to obtain a valid source file that can be successfully analyzed by the C parser. This requires that we have source files that are ready to go through the compiler. To get the source files in the open source projects to this state requires a bit of work. This chapter describes the infrastructure we built to facilitate our work.

The first task is to extract a valid source tree from the CVS repository. The difficulty here is to ensure that the source tree we extract matches exactly what the developer saw in the source tree when the CVS commit was performed. Once the source tree has been extracted, the source code needs to be configured. This is usually done through a script that is part of the source tree. During this step, a number of files are customized to work correctly with the specific operating system and software development tools, such as compilers, installed on the particular machine on which the source code is being built. This step is necessary since many of these tools could be installed in a variety of locations. Also, the open source projects we are studying can be built on a number of different operating systems which may have various system header files in different locations. Further, if the source code relies on having a particular version of a software development tool to compile the source code, this is often the step where that check is performed. This step typically produces Makefiles and sometimes is used to produce source code

header files. Finally, we can run the Makefiles to determine how the compiler is invoked for each source file. In this step, it is necessary to determine the command line options given to the compile to compile each source file. These options may specify which directories contain the source code header files needed to compile each source file. Our parsing tool will need to know where to look for these files. The following sections discuss these steps in greater detail.

3.1 Interaction With CVS

The following sub-sections detail how a valid source tree is extracted from CVS, how we manage the data stored in CVS and some shortcomings of CVS that have presented challenges to our work.

3.1.1 CVS Transactions

CVS is a version control system used to track changes in a set of files [18]. It is widely used by software development projects to manage source code and provide a way to allow multiple developers to update the same source code concurrently. All the changes made to the source code are store on a file by file basis. While CVS does allow the user to change multiple files at once, the changes are actually stored at the file granularity. For each file, CVS stores a set of change records that describe the contents of the file. Each change record contains a set of changes applied to the file, a time stamp, the username of the author of the changes and a commit message provided by the author that can be used to describe the contained changes. Since there is no explicit information stored to describe which files were updated together, we need to rebuild that information based on the change records.

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.

A series of CVS 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.

We have implemented a variation of the sliding window algorithm described by Zimmermann and Weissgerber [75] to rebuild the CVS transactions. However, unlike the Zimmermann and Weissgerber algorithm, we do not rely on commit messages when rebuilding a CVS transaction since a programmer may 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.

Each file that is updated in the CVS transaction is updated exactly once in that transaction and each update performed on a file in the transaction will be done by the same author. Our implementation of the algorithm first sorts the entire list of CVS commits by the timestamp in the change record. The first CVS transaction is built by

finding all changes within a window of time made by the author of the first commit. The beginning of the window is then slid to the commit with the most recent timestamp. Any commits within the window made by the same author are added to the CVS transaction and the window is slid again. This process continues until the window does not contain any commits by the same author. The size of the window is described below.

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 usage styles differ so much, even if CVS did record the transactions, 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. For the Apache httpd software repository, the decrease in the number of transaction found was substantial between the 30-second and 180-second windows, a reduction of 1,189 transactions (8% 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 our work. Table 1 lists these values.

Window size in seconds	Reduction in number of CVS Transactions from previous window size
30	--
180	1,189
240	313
300	212
360	198

Table 1: Affect of Window Size on Number of CVS Transactions found

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.

3.1.2 CVS Data in a Database

CVS is implemented as a layer on top of an earlier revision control system called RCS [62]. While the data is stored in the 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 [68]. 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 [20]. The schema we have used is a modified version of the schema described by Zimmermann [75]. The changes we made were to store information regarding directories together with the information stored for files and to omit CVS branch information.

In our schema, we have a table containing information about each file including the name, directory name and a file key. Another table contains the revision data for each file. Each row in this table represents a commit to a single file. The row contains a revision key (the CVS minor number), the commit message, the timestamp of the commit, the author's username, the full text of the particular version of the file and the file key to link back to the file table. A separate table represents the CVS

transactions, consisting of a series of rows each containing the transaction key, a file key and a revision key. Two tables to represent the tag information are also built. The first contains a text field that contains the tag and a tag key. The second table contains the tag key as well as a file and revision key to map the tag to a specific version of a file. In CVS a particular file and version may be tagged more than once and one tag may be applied to multiple files. Two auxiliary tables are created, one containing the transaction key and the minimum timestamp of all revisions in that transaction and a similar table containing the maximum timestamp. These tables are provided for performance reasons to provide quick access to the boundary times of the CVS transaction.

With regards to the directory information, we believe this more accurately represents the way CVS stores information (see section 3.1.3 describing 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, with cheap storage devices the cost is small, and it makes recreating a source tree and searching the text of the revisions faster and 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.

3.1.3 Moving a File or Directory in CVS

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 approaches brings on a number of problems very similar to those discussed for moving a file. In both case, a moved file or directory, corrupts the source tree and prevents older snapshots form being built correctly.

3.2 Building Yesterday's Source Tree Today

One of the key engineering challenges faced when mining the software repository of an open source project is building older versions of the source code on newer versions of operating systems using newer versions of software support tools.

3.2.1 Running configure

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.

Running configure on many of the older source trees stored in the repository proved to be a challenge. In the Apache httpd project especially, we had a difficult time configuring old source trees (from the first three years of the revision history, 1996 through the first half of 1999). One cause of this was the fact that we were doing our analysis on a Linux machine running a 2.4 kernel. In the configure files for older source trees the scripts look for a Linux kernel version of 2.0 or 2.2 explicitly. Any other kernel versions caused the configure scripts to terminate with an error

message. To resolve this problem, we edited by hand a small number of configure files to include a check for the 2.4 Linux kernel.

Another challenge involved the tool autoconf. A particular, older, version of autoconf (2.52) contained a bug that necessitated the autoconf scripts to be rewritten to contain a workaround. Unfortunately, this workaround was incompatible with later versions of autoconf that did not exhibit this bug. Once again, we had to update these scripts by hand to have the source code configure successfully.

Finally, to run the configure scripts on each source tree can take up to 3 or 4 minutes. This makes the computational cost of evaluating each CVS commit extremely high. To save time, we ran configure once per CVS transaction and used gcc to produce a preprocessed version of each source file changed in that transaction. This preprocessed file was then stored in a database. Our analysis tools operated on these preprocessed files without the need to run the configure commands. This is discussed in greater detail in the following sections.

3.2.2 Preprocessing with GCC

The tools we have developed to analyze the source files parse the file in a similar manner to a complier. This requires that all the header files and other dependencies be in place. To do this, the configure scripts need to be run. As stated above, running the configure scripts for a source tree can be very time consuming and so we want to run these scripts as few times as possible. To facilitate this, we have used gcc (using the –E option) following configuration to produce a preprocessed source file for each version of each file in the repository. This preprocessed file depends on no external files, the relevant bits of the header files are placed inline in the file and any *#define*

macros are fully expanded inline. Additionally, the file is annotated with the original line numbering information so that source lines in the new file can be mapped back to those in the original file. This allows us to build and configure the source tree just once for each CVS transaction. Our tools operate on these preprocessed files, and so we are free to rerun our tools, or run new tools, as many times as necessary while only running configure for the source tree once. Storing the preprocessed files saves a tremendous amount of time. To preprocess the source files, our tool needs to have access to the command line options given to the complier to specify include directories as well as defined values (-I and –D options to gcc, respectively). The Makefile can be run to generate this information.

We wanted to avoid actually compiling as much of the source code as possible since the software projects we studied can take over 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. We can parse this output to determine what options to give our invocation of *gcc –E*. 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 (or in the –I command line options) may reflect that. Both our scheme and the one outlined by Ferenc, et al., [27] 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. Ferenc, et al., proposes dealing with Makefiles that move files around 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.

3.2.3 Versions of Software Support Tools

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 is

vital to configuring and building a source tree and seems like a natural fit for the version control system.

3.2.4 Line Number Mapping

To determine the changes that have been made between two versions of a source file, we need to know how the source lines in the previous version map onto the source lines in the subsequent version of the file. Changes to the source file, including adding and removing lines or moving a function to a new location in the file, may cause the same line of source code to be located at different line numbers in the two versions of the file. Recognizing this line number mapping allows our tools to correctly identify new instances of properties and, more importantly, detect when an instance of a property that exists in both versions of the file has simply moved to a new source line.

We determine a mapping between the source lines of successive versions of each file. Our work on this was based on the line number matching scheme described by Spacco, et al. [65]. The line number mapping is done on a per-function basis. For this we use the normalized edit distance (true edit distance / maximum possible edit distance between the two lines based on the length of each line) between the lines of source code in the function in each version of the file to provide weights for the edges of a bipartite graph. The true edit distance is the minimum number of operations needed to transform one string into another. The maximum possible edit distance is the edit distance between two strings if those strings contained no common characters in the same location. We then find a minimum weight bipartite matching to determine the best mapping between the two versions of the function.

We also set a maximum normalized edit distance so that if a line matches a line in the previous version, but the edit distance between these two versions is greater than the maximum normalized edit distance, the mapping is considered to not exist. This is done because the matching algorithm will match as many lines between the two file versions as possible. The maximum normalized edit distance we used was 0.31, which was chosen after reviewing the matching results from a number of revisions of different files.

3.3 Database for Results

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.

3.4 Computation Costs

In order to mine all these CVS transactions we used a 64-node (each node containing 2 CPUS) Linux cluster managed by the Portable Batch Scheduler (PBS) [58]. Each node hosted a copy of the database containing the CVS data and the

results database and one node hosted the master database. That allowed almost all database transactions to happen locally to reduce the amount of traffic on the network and to not overburden the master database. The master database was used to store the final results and also to distribute blocks of unique keys to the nodes. These unique keys allow the data from all the nodes to be combined into a single database. After a job was completed, the results produced by each node were gathered up and stored in the master database. 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 mid-level 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.

To build the database for each of the open source projects we studied took quite some time. Inserting the files into the database from CVS took around 12 hours for each project. Building the CVS transactions for each project took another hour. The most time consuming part is checking out each transaction, running the configure scripts and producing the gcc –E output for each file changed in that transaction to store back into the database. Building the line number mapping also took considerable time, around 12 hours for each open source project. The entire process took around 48 hours using 25 nodes on the cluster. This process only had to be done once for each software repository.

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 2 minutes (note that because we operate on preprocessed source files we do no include the cost of running configure on the source tree in this number). Our bug finding tools take about as long as a compiler to analyze a source file and never took more than a few tens of seconds on any single source file we checked. In total, each of the open source software projects we studied took around 48 hours to mine using 31 nodes on the cluster. This includes the time to mine the source code as well as to analyze the data. Analyzing the data involves determining the new instances of the properties and generating the statistics that are used to apply the data to the results of the bug finding tools we produced. The student data could be fully mined in under two hours using two nodes.

Chapter 4: Function Return Value Checker

In this chapter, 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 chapter 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 software. We have done this by a manual inspection of the bug database and source code repository of the Apache web server [69]. 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. This particular bug was found frequently in our manual inspection. 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.

4.1 Preliminary Mining of CVS

The first step in this investigation was to review the historical data for the Apache web server, httpd [5], 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 data about fixed bugs exists and how easy it is to identify such data. We looked at both the bug database and the CVS commit messages in the repository.

To start our investigation, we reviewed the bug database for the Apache web server, httpd. 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. To directly tie a bug report to a CVS commit we look for a comment in the bug report that states a particular version of a file (or files) was created to fix the bug. We were only able to tie 24% of the bug reports marked as fixed directly back to a code change. While a developer could 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 2 contains a breakdown of the bugs we were able to classify from the bug database. The first column contains the classification of the bug, the next column

contains the raw number of instances of this bug that was identified, the last column contains the percent of total bugs that we inspected that were classified as such. Most of the bugs we classified (34 of 200) were logic errors or feature requests. Feature requests are a new feature for the software or porting a feature to a new platform. We categorize bugs where the code is correctly written to perform the wrong thing 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). The NULL pointer check bugs are instances where a pointer was not checked for NULL before being dereferenced. The Return Value check bugs are bugs that do not test the return value of a function before using it. Uninitialized variable bugs arise from a value being read before being defined. Error branch confusion denotes bugs that arise from swapping code between the success and failure case after a conditional. External bugs are all bugs that denote the failure of some third party system and did not require a source code change to Apache httpd. System specific bugs are violations of a system specific rule that the source code must adhere to. Calling a mutex locking function before accessing a particular piece of shared data is an example of this.

NULL pointer check	3	1.5%
Return Value check	4	2.0%
Logic Errors/Feature Request	34	17.0%
Uninitialized Variable Errors	1	0.5%
Error Branch Confusion	2	1.0%
External Bugs (OS or other software failed)	2	1.0%
System specifc pattern	1	0.5%
No identified code change	153	76.0%

Table 2: Bugs Identified in the Apache Bug Database

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 encounter few of these bugs.

In order to understand what types of bugs are being committed to the software repository, but are not being listed in the bug database, 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 reported in the bug database as possible. Moreover, we only looked at files that had a larger number of commits to them, 50 or more. Table 3 shows the breakdown of the bugs we were able to identify from the CVS repository. The first column contains the classification, the second column contains the raw number of instances of this type of bug identified, and the final column lists the percent of all inspected bugs classified as such. The new classifications not listed in Table 2 are “Failure to set value of a pointer parameter”, “Error caused by if conditional short circuiting”, and “Loop iterator increment error”. The pointer parameter bugs denote passing an uninitialized pointer to a function. The conditional short circuiting bugs are cases where short circuiting in a conditional statement would prevent a necessary function from being invoked. The loop integrator bugs were cases where the loop iterator was incorrectly incremented, either

by not incrementing them or using the wrong increment value. 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.

NULL pointer check	28	40.6%
Return Value Check	29	42.0%
Uninitialized Variable Errors	3	4.3%
Failure to set value of pointer parameter	1	1.4%
Feature Request	1	1.4%
Error caused by if conditional short circuiting	1	1.4%
Loop iterator increment error	3	4.3%
System specific pattern	3	4.3%

Table 3: Bugs found the Apache Software Repository

Our preliminary, manual inspection of the data gave us two important insights. First, we determined that automatically mining the source code changes in the CVS repository [18], 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 with the corresponding source code changes. Secondly, we discovered, by examining the software repository, a list of bug types that were commonly fixed in the software project we studied. 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.

4.2 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. We do not include function pointers in this list. 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.

4.2.1 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 by building a def-use chain for that value. If the variable, during its first use after being defined, 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 [42]; 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. The function return value checker implemented in lint also only flags functions whose return value is ignored. The analysis performed by our tool, ensuring the return value is tested in a conditional, is more sophisticated.

Our checker places 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 4 shows the complete list of categories of warnings our checker reports and provides a brief description.

Warning Type	Warning Description
Ignored	Return value not stored in a variable and not used directly
Argument	Return value passed as an argument to a function
NULL dereference	Return value dereferenced
Calculation	Return value used in an arithmetic calculation
Stored, Unused	Return value is stored in a variable but never used
Unused on Path	Return value is stored but unused on some path out of the function
Stored, Untested	Return value stored and used but not tested

Table 4: Warning Types

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.

4.2.2 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 (within that function) 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 that could be investigated 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.

4.2.3 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.

4.3 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 multi-person, multi-site effort and the resulting software is in daily use by many people.

4.3.1 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. Note that the Naïve Ranking is a ranking we created to determine if using the historical information is better than merely using information from the current version of the code.

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 their 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.

4.3.2 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, and 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, for a number of reasons. Some CVS commits would not configure correctly, for reasons discussed in section 3.1. Some files contained C constructs that the parser [61] we used could not handle, most notably having a variable number of arguments to a *#define* macro. Also, the parser does not yet fully support parsing of GNU extensions to the C language [66]. A small number of commits also had compile errors where a file with a syntax error was checked in to the repository.

4.3.3 Special Considerations

The Apache web server source code is interesting in a number of ways. First, the code is divided into several 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

[1] Our study was confined to the 2.0 branch.

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 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 was located in a different directory than it is today. See Chapter 3 for a discussion as to why this is a problem and other shortcomings of CVS.

4.3.4 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%) have their return value checked

100% of the time in the current CVS snapshot and so are involved in no warnings. For comparison, 56% of all functions (1,001) had their return value checked 100% 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% of the time in the contemporary context. We chose 11% as our lower bound after inspecting warnings produced by functions down to 1%. We observed that all warnings below the 11% mark were false positives.

		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%

Table 5: Warnings and Likely Bugs for Apache

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% for this chunk of our results (functions flagged with a CVS bug fix). See Table 5 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 5 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% 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%. See Table 5 for the breakdown of these results. Table 6 contains a breakdown of the warnings we inspected by category (see section 4.2.1 for a description of these categories). The NULL dereference warnings have a fairly low false positive rate. Other types of warnings, especially Unused on Path, have a high false positive rate across the rankings.

Warning Type	CVS bug fix flagged functions				Non-CVS bug fix flagged functions	
	Checked (50%-99%]		Checked [11%-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 6: Warnings Reported for Apache

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%. The remaining 6,151 warnings marked by our checker are produced by functions whose return value is checked 50% of the time ore less. These functions

have also not been flagged with a bug fix in the CVS history. The functions flagged with a bug fix in CVS and that have a contemporary context of 50% or less produce warnings with a reasonable false positive rate. We surmise that these functions are often misused but the bug fixes are evidence that their contemporary context is on the rise to above 50%. The some of the functions that produce the 6,151 warnings may also be often misused. However, without a history of bug fixes we expect that identifying these functions would be difficult and lead to a very high false positive rate. Evaluating 6,151 warnings also raises practical considerations.

A false positive rate closer to 50% 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 4.4.2.

4.3.5 Wine Case Study

We conducted a second case study of our checker on the Wine source code [72]. This is another open source project with an extensive CVS history. We mined over 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. This code base contains calls to approximately 16,000 unique functions. 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 Chapter 3. Some files contained C constructs that our parser could not handle, most notably having a large multidimensional array initialized in a declaration.

4.3.6 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 Chapter 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 email 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.

4.3.7 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 are flagged with at least one 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% of the time in the current CVS snapshot (34%) and so are involved in no warnings. For comparison, 37% of all functions (4,404) had their return value checked 100% 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% of the time in the contemporary context.

		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 7: Warnings and Likely Bugs for Wine

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 gives a false positive rate of 67% for this chunk of our results (functions flagged with a CVS bug fix). See Table 7 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% 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%. Overall we inspected

2,315 warning reports and found 546 that we believe are suspicious and should be marked as a bug. This gives an overall false positive rate of 76%. Table 7 shows the breakdown of these results. Table 8 contains a breakdown of the warnings we inspected by category.

Warning Type	CVS bug fix flagged functions				Non-CVS bug fix flagged functions	
	Checked (50%-99%]		Checked [11%-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

Table 8: Warnings Reported for Wine

4.4 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.

4.4.1 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% 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.

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

Figure 1: Example Return Value Check Bug

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. This may indicate a misunderstanding of how the string searching functions operate. This may also point to the fact that the developer is making the assumption that the strings they handle will be well-formed and contain the data being sought, thereby ignoring the failure case for these functions. 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 Figure 1. If the function `ap_server_root_relative` returns the value NULL the code snippet will cause a segmentation fault. This may indicate two assumptions by the developer. First, the data structure has been built and populated correctly. Secondly, the data being sought will be found. 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% 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 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.

Again, this set of functions seems to indicate the developers are willing to assume that global data structures are created correctly and contain the data being sought. However, in this group we also see functions that perform complex operations that could fail. Developers seem to ignore the possibility of failure and produce code that assumes no errors in the data to be manipulated.

4.4.2 Where are the bugs in the rankings?

In analyzing our results it is important to look at where the likely bugs are found within our ranking of warnings. Recall that 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 Venn diagram in Figure 2 shows how the HistoryAware Ranking interacts with this notion of checking only warnings produced by functions with a contemporary context of more than 50% 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. Figure 3 shows the same thing for the Wine results. Figures 10 and 11 examine the false positive rates for the subsets of warnings produced by using cutoff rates other than 50%.

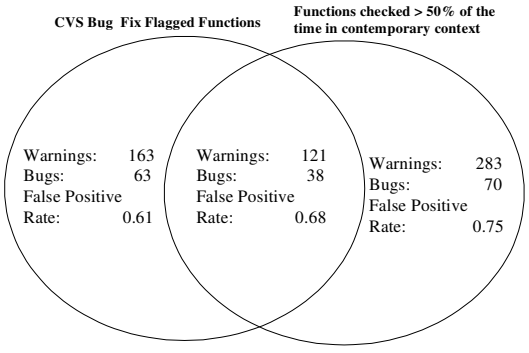


Figure 2: Division of Warnings, Apache

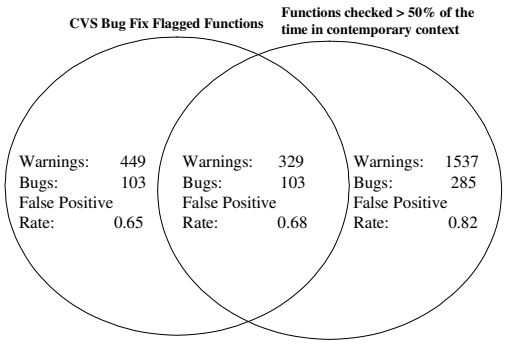


Figure 3: Division of Warnings, Wine

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.

4.4.3 Statistical Significance

In order to evaluate the statistical significance of 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 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% 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% 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% confidence. In fact, the Wine data was statistically significant at the 0.001 level. Table 9 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 parentheses explain how the numbers were derived and are from the data in

Table 5: Warnings and Likely Bugs for Apache

.

	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

Table 9: Apache Chi-square Calculation

4.4.4 Precision

In Figure 4 and Figure 6 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. The y-axis in these graphs measure precision and the x-axis is the number of inspected warnings. The warnings are inspected in ranked order, either by the HistoryAware or Naïve Ranking. Figure 5 and Figure 7 provide a more detailed view of the left side of the graphs, representing the first warnings inspected, from Figure 4 and Figure 6, 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.

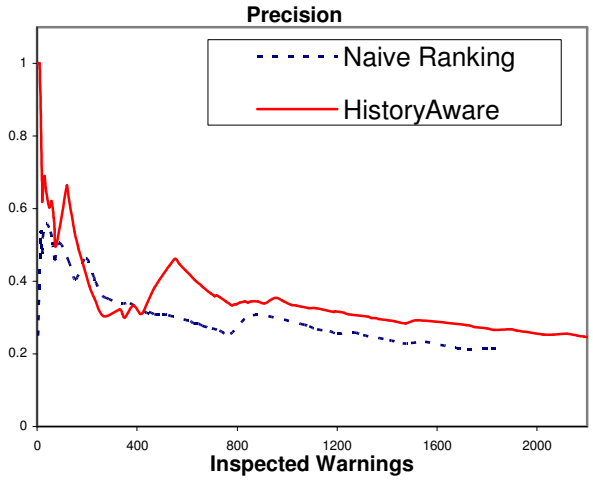


Figure 4: Precision in the Wine Case Study

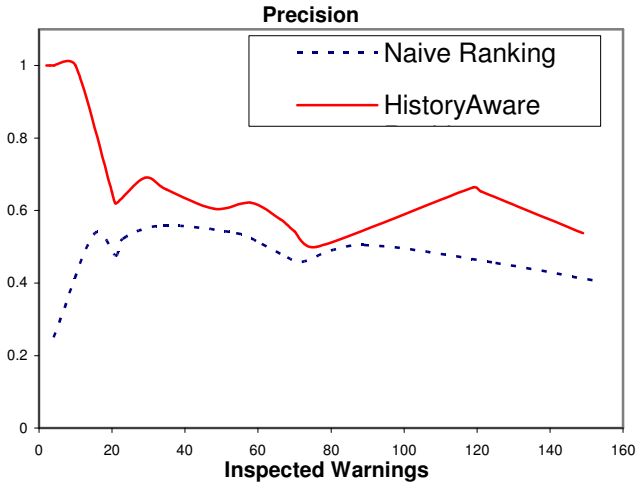


Figure 5: Precision in the Wine Case Study, detail

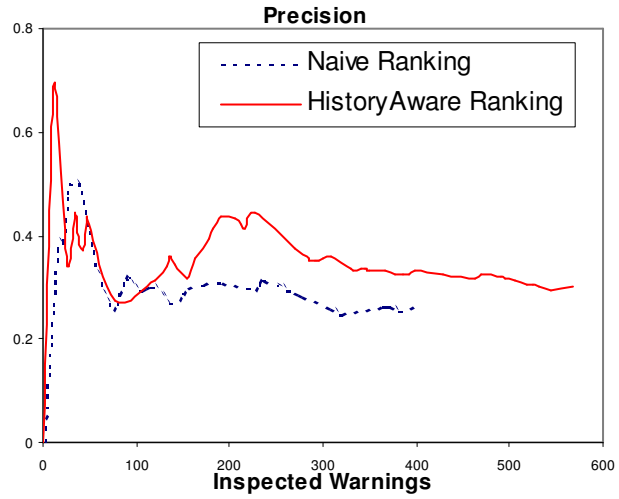


Figure 6: Precision in the Apache Case Study

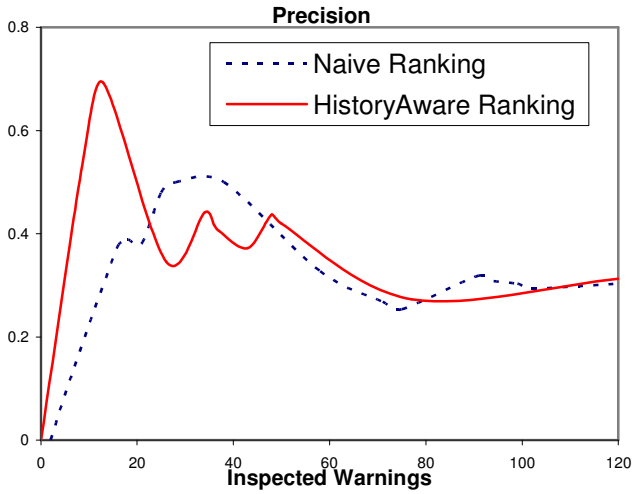


Figure 7: Precision in the Apache Case Study, detail

Figure 4 and Figure 6 show the precision over warnings inspected in two different lists of warnings. Figure 4 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. Figure 6 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 Figure 4, 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 Figure 7, for the top 50 warnings the precision for the Naïve Ranking is 0.32, while the precision for the HistoryAware Ranking is 0.42.

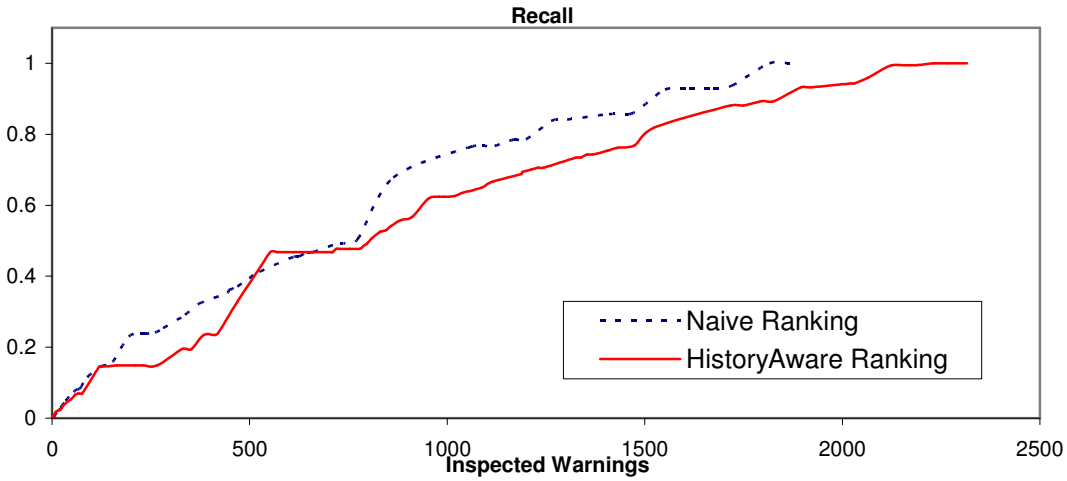


Figure 8: Recall in the Wine Case Study

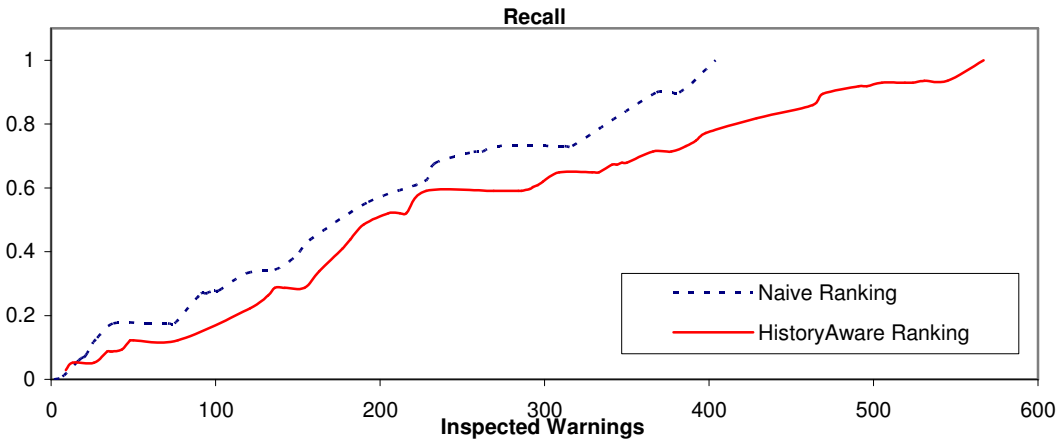


Figure 9: Recall in Apache Case Study

4.4.5 Recall

Figure 8 and Figure 9 plot the measure of *recall* of the same two sets of warnings for Wine and the Apache web server, respectively. The y-axis is recall and the x-axis is the number of inspected warnings. The warnings are ranked by either the HistoryAware or Naïve Ranking. 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 precision, or density of likely bugs, is higher for the HistoryAware Ranking.

4.4.6 Cumulative False Positive Rate

Figure 10 and Figure 11 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. The x-axis is similar to the previous graphs, however here the warnings are sorted solely

by contemporary context. 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 – NonCVS 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 NonCVS 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.

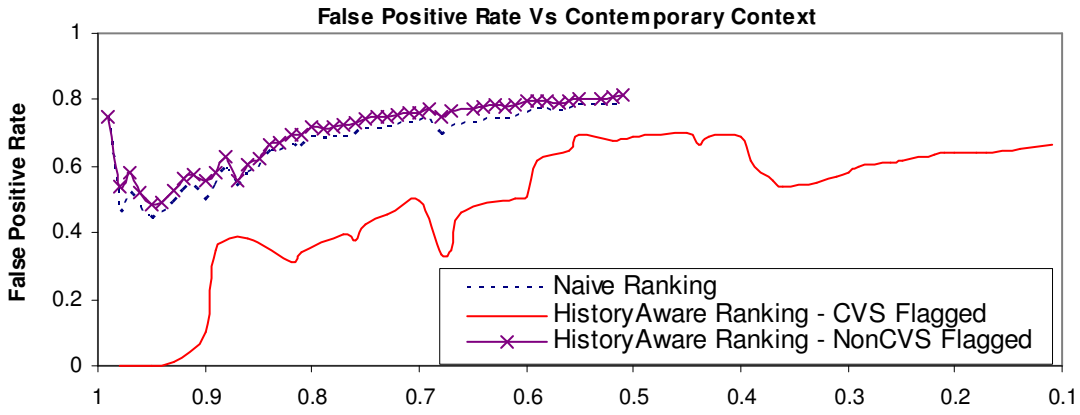


Figure 10: False Positive Analysis in the Wine Case Study

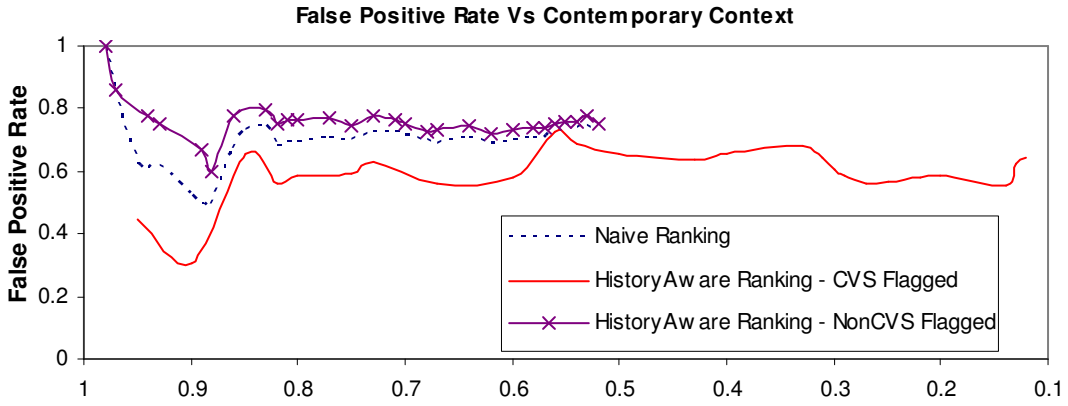


Figure 11: False Positive Analysis in the Apache Case Study

While the false positive rate for our HistoryAware ranking remains high, over 50%, this is a considerable improvement over the Naïve Ranking of warnings produced by our tool and over traditional, unsorted compiler warnings and tools such

[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.

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.

4.5 Threats to Validity

This section discusses a number of threats to the validity of our experiments. Only one person (the author of this dissertation) inspected the large number of warning messages produced by our tool. That person 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. We would also need to produce an artificial history of bug fixes to mine for historical data. 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`.

4.6 Computation Costs

A general overview of the computational cost of mining these software repositories is discussed in Section 3.4. Analyzing the latest version of the source code for missing return value checks takes about as long as compiling the code. Note that as with mining the source code, the code tree does need to be configured before the bug finding tool runs. Sorting the warnings produced by this analysis using the historical information takes a trivial amount of time. This consists of a database query per function that causes a warning and a sort operation.

4.7 Summary

In this chapter 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.

The key finding in this chapter is that the historical information adds benefit to the static analysis tool. Further, the benefit added is greater than if only the current version of the source code is mined for data. This shows that the property that we have examined here, how to handle the return value from a function, can be mined

from the source code repository. The effort required to mine the source code repository has been shown to produce a static analysis tool.

From a preliminary manual 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 work in this chapter.

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 precision of the rankings produced by our technique was consistently higher then that of the naïve ranking.

We have used an arbitrary threshold of 50% contemporary context to determine which warnings to inspect where the function involved is not flagged with a bug fix in CVS. This threshold value may not be the best choice. The investigation of the warnings below this threshold, or some sampling of the warnings, would produce a

more definitive answer. The historical information mined from the repository is represented as a single metric, whether the function flagged with a bug fix or not. Using a more comprehensive metric, like the number of times the function is involved in a bug fix may provide different results and is worth more study. Also, with the historical data trends could be identified. Seeing a large number of bug fixes to a function in a short period of time may indicate that the function's implementation has changed and therefore warnings produced by calls to it should be ranked higher.

Chapter 5: Function Usage Pattern Miner

Static analysis of source code has been used very successfully to locate bugs in software. One of the most successful applications of static analysis to find bugs has been tools that look for violations of system-specific rules in the source code. Source code must adhere to a large number of rules that describe how data should be handled, how to interact with objects or APIs and how to use functions safely. Violations of these system-specific rules are often a source of error [49].

The difficulty with these rules is that they are implicit and constantly changing. As the source code changes, new rules are added and old rules are removed. When functions are added to an API a new set of rules is created that describe how these functions are to be used. It is challenging for the developers of a geographically distributed project to keep track of the rules the code must follow. This task is complicated by the fact that many of these rules are not documented as they are created, or are only documented in a CVS commit message or an email on a developer mailing list.

This leaves the project to rely on developers learning these rules in a number of unsatisfactory ways. For example, the project may rely on senior developers relating the rules that they know to new developers or developers searching CVS commit messages and mailing lists when they have a question or developers inferring them from the code. New developers are not the only ones to suffer. Senior developers need to keep up with the rules being added and removed from the source code as well. In a large project, it is very likely that a developer will primarily work on a

particular piece of the code and understand the implicit rules describing that code. However, when the developer needs to interface with a different area of the code, the implicit rules that describe its usage may not be as well understood.

In this chapter we investigate if it is possible to extract these implicit rules from source code repositories. We describe a method for mining function usage patterns from the revision history and three tools that use the mined function usage patterns. The first tool looks for violations of function usage patterns, the second tool documents an API with a series of function usage patterns, and the final tool examines the source code to identify the emergence of a new function usage pattern.

For each function usage pattern mined from the repository we calculate a number of statistics that represent how new instances of that pattern are created in the code. These statistics are used to select which function usage patterns are likely to be valid and are subsequently used by our tools. The threshold values used to select the function usage patterns have been determined experimentally by examining the results produced by a much larger selection of function usage patterns. The thresholds were chosen to reflect the area in the data that provided high benefit to the developer. In effect, these thresholds act as knobs used to tune the performance of the tools. While other software projects may need different threshold values we believe the ones used here are a good starting point for analysis. Each of the case studies in this work used the same threshold values. The values were not tuned to each software project separately.

5.1 Function Usage Pattern Miner

The system-specific rules that we study in this chapter are function usage patterns. We want to determine how functions are invoked with respect to each other. Experience suggests that there are sets of functions that need to be invoked together to perform a larger conceptual goal. These functions may operate on common data, provide error recovery functionality or perform some type of pair-wise functionality like lock/unlock. The specific function usage pattern template we are looking for is the *called after pattern*. The called after relation occurs when function X is called after function Y in the body of some function Z. Figure 12 provides an example of the called after pattern. Instances of these patterns in a software project build up a set of relationships between functions. We define an instance of a function usage pattern as a set of two particular function call sites, within the same function, such that the pattern template is satisfied.

```
HDC hdc = BeginPaint( hwnd, &ps );
if( hdc )
    DrawIcon( hdc, x, y, hIcon );
EndPaint( hwnd, &ps );
```

Figure 12: Called After Pattern

We have implemented a tool that performs a static analysis of the source code to produce a list of instances of function usage patterns for each function in the code. Our tool reports every pair of function calls that exist on each path through a function. This analysis is intra-procedural and produces a large number of instances of patterns from each function. This gives us the freedom to put off making decisions on how to filter the data until later in the process. This is important as retrieving the data from the software repository and generating our results is the most computationally

expensive aspect of this work. The tool we have implemented accepts C source code and is based on the Edison Design Group C parser [22].

We use the framework described in chapter 3 to manage the data from the CVS repository and the results produced by our tool. This allows us to implement our mining algorithms in terms of database queries. The function usage patterns are stored across two tables in the database. The first table is keyed by the tuple: file, revision, calling function, function one and function two. This table also contains a RelationID used to uniquely identify instances of this pattern in this function in the particular version of a file. All instances of this function usage pattern found in this function in this file version have the same RelationID. The second table contains the line number and data flow information for each specific instance of a pattern. This table is keyed by a RelationID and a RelationIndex. The RelationID and RelationIndex pairing uniquely represents one instance of a particular function usage pattern in the history. Each row in this table contains a RelationID, RelationIndex, the line number information for each function and the data flow information for the instance. Separating this data into two tables allows quicker access to the first table as it contains about one quarter the number of rows in the second table. Most of the queries made to the database are based on string matching to the first table and then a quick lookup through the RelationID key to the second table.

5.1.1 Control Flow Graphs

The tool parses the source file and builds a control flow graph (CFG) for each function in the file. A pass is made over the CFG to determine which basic blocks contain function call sites. For each basic block that contains a function call, the tool

determines the set of basic blocks that are reachable from that basic block. The reachability analysis is done by looking at the CFG as a directed graph. Data flow analysis is not used in this stage. Function calls in each of the basic blocks in this set are paired with the function calls in the original basic block and are output as an instance of a function usage pattern. For each instance of a pattern, the tool records the names of each called function, the line and column numbers of the call sites and the name of the enclosing function, as well as the data flow information detailed below.

5.1.2 Data Flow Information

After determining the set of instances of function usage patterns within a particular function, a data flow analysis is performed to determine if the invocation of the functions touch the same data. This is done to provide more local context information describing the instance of the function usage pattern. The data flow information is used to select patterns that are more likely to be valid patterns and reduce the number of patterns our tools consider. There are three ways the invocations can touch the same data. Each of these relationships are identified by doing reachability analysis on the variables involved. First, the invocations of the functions can be given the same variable as a parameter. If the variable may have the same definition for both function invocations then there is a data flow relationship between the two function calls. Second, the second function invocation can take a value returned by the first function invocation as a parameter. This is an application of a def-use chain. If the first function invocation defines a value and that definition may reach the second function invocation a data flow relationship exists. Finally, the

function invocations can both update that same variable. An example of this is to have the first function call allocate a struct and the second function call's return value stored in that struct. Examples of these data flow events are show in figures 13-15.

Note that we are doing a flow-sensitive data flow analysis to determine these facts and not merely string matching to identify common variables between the two function invocations. Our tool does not, however, do pointer aliasing. While just using the same variable to pass a value to each function invocation may, in some cases, indicate a relationship between the two functions, we saw many cases where a temporary variable was redefined and reused to supply values to unrelated function invocations. For this reason, we do the full data flow analyses listed above. The information we generate with the data flow analysis is used to determine which function pairs are likely meaningful function usage patterns and to reduce the number of function usage patterns our tools need to consider. The use of this information will be discussed in greater detail in Section 5.4.

```
HDC hdc = BeginPaint( hwnd, &ps );
if( hdc )
    DrawIcon( hdc, x, y, hIcon );
EndPaint( hwnd, &ps );
```

Figure 13: Data Flow, Same Parameter

```
HDC hdc = BeginPaint( hwnd, &ps );
if( hdc )
   DrawIcon( hdc, x, y, hIcon );
EndPaint( hwnd, &ps );
```

Figure 14: Data Flow, Produce/Consume

```
p = malloc();
p->value = getX();
```

Figure 15: Data Flow, Update Same Variable

5.2 Mining the Repository

When mining the software repository we are looking for a new instance of a function usage pattern in a revision of a file, where that instance of the pattern did not exist in the revision immediately prior. We are interested in how and which new instances of patterns enter the code as the result of a code change. Once we have the list of new instances that have been added to the code, we calculate a number of statistics to measure how likely a particular function usage pattern is to be a meaningful rule the source code needs to follow. Statistics that describe how the function usage pattern has entered the source code are also calculated. The following subsections detail how we determine what new instances of function usage patterns exist and how we use this information to determine a set of rules that describe how the functions within the source code should be invoked.

5.2.1 Finding New Instances of Patterns

We run our tool on each version of each file in the CVS repository to determine the function usage patterns present in each version. To determine which instances of function usage patterns have been added to a file we examine the instances of the patterns found in versions *n-1* and *n* of a file. Due to syntax errors and configuration problems not every version of a file may be syntactically meaningful and so we may not be able to analyze it. We confine our analysis to successive, compilable versions

of a file. Therefore version *n-1* of a file is actually the most recent version of the file prior to version *n* that is compilable.

Our mining tool looks at every instance of a pattern in version *n* of a file and determines, based on the line number mapping, whether or not that instance appeared in version *n-1* of the file. If no line number mapping is found for the line that contains a function call involved in an instance of a pattern in version *n* of a file, then we note that function call has been added and this is a new instance of a pattern. If a line number mapping is found for the line that contains a function call, the line in version *n-1* is examined to determine if it contains the same number of calls to the particular function involved in the function usage pattern as the line in version *n* does. If it contains fewer invocations of the particular function, then this function call is determined to have been added. It is almost always the case, however, that adding a function call to a line of source code changes that line significantly enough that it does not map back to a line in the previous version of the file.

We store information about the new instances of patterns in a database. This allows us to use database queries to mine information from the CVS history quickly and easily. For each new instance we track whether each of the two function calls were added to the source code to create the new instance. How the new instance is added to the code can reveal the degree to which the pattern has caused problems to the developers in the past. Patterns that are often added to the source code by adding both function calls in the same commit denote patterns that are likely to be meaningful patterns that the developer has a good understanding of how to use. Patterns that are often created by adding only one of the function calls to the source

code may denote patterns that are poorly understood (developers went back and fixed the incomplete pattern) or that arose out of a refactoring (e.g., the addition of a function to the source code).

5.2.2 Mining Rules from the Repository

We generate, for each pattern (`foo -> bar, bar` called after `foo`) mined from the repository, a number of statistics to determine how likely a pattern is meaningful, how likely it is to be buggy, and how it has been created in the past. We use these statistics to determine which function usage patterns we will later use to look for bugs and build APIs. The following statistics are calculated by only looking at new instances of a pattern where a data flow relationship is present, unless otherwise noted.

The confidence statistics detailed below are based on the value of confidence calculated for association rules produced by using the Apriori algorithm [2]. Each of these statistics measure how often, when a particular event occurs (create a new instance of a function usage pattern, add a function call), does the resulting code exhibit a particular property (data flow relationship, new instance of a function usage pattern).

The statistic *DataFlowConfidence* calculates how often, when an instance of the function usage pattern is created, that instance exhibits a data flow relationship of any kind. This is the primary statistic we use to determine worthwhile function usage patterns. We will be interested in function usage patterns that have a *DataFlowConfidence* of 0.8 or greater.

```
Void foo(){                    Void foo(){
                                      open();
      . . .                           . . .
      close();                        close();
}                              }
```

Figure 16: Change that Results in an Increase of ConfFirst

The statistic *ConfFirst* calculates how often, when the first function call in a relationship (`foo` in the example above) is added to the source code, is the relationship `foo -> bar` created. An example of a source code change is Figure 16. On the left is the original code, on the right the code after the change. After `open` is added the value of *ConfFirst* for `open -> close` will increase. We calculate this by taking the number of times a new instance of the relationship is added to the source code by adding a call to `foo` and dividing by how many times a function call to `foo` is added to the source code. This gives us a value in the range of 0 to 1, inclusive. For this measure, the relationship may be created by adding either a function call to `foo` or a function call to both `foo` and `bar`. Functions may be added to the source code at different times (sometimes at radically different times) but still form a usage pattern. When this happens, the value of *ConfFirst* (and the similar value *ConfSecond* which we will describe in the following paragraphs) can be wildly skewed toward zero. All of the additions of calls to `foo` (if it is the first function to be defined in the source code) before `bar` is defined will not create an instance of the function usage pattern `foo -> bar`. But, when this is a valid usage pattern, all the additions of `foo` after `bar` is created will create an instance of the function usage pattern. This disjoint creation of functions and subsequent creation of

function usage patterns is exactly the type of hard to learn, implicit rule that our work is trying to identify. For that reason, we only look at additions of calls to the function `foo` in CVS transactions after both the function `foo` and the function `bar` are defined in the source code. Any additions of calls to `foo` before `bar` exists as a function in the source code are not counted in the statistics for `foo -> bar`.

The statistic *ConfSecond* calculates a similar metric to *ConfFirst*, but for the second function call in the function usage pattern. It is, again, only calculated from additions of the function `bar` after both `bar` and `foo` are defined in the source code.

These two measures give some indication of how tightly coupled the function calls are. For example, in the Wine source code we found the pattern `WCEL_DeleteString` *is called after* `WCEL_SaveYank` to have a *ConfFirst* measure of 1.0 but a *ConfSecond* measure of 0.25. Since `WCEL_SaveYank` (save a string copied from the console) needs to have `WCEL_DeleteString` called afterwards every time (to avoid a memory leak) but `WCEL_DeleteString` may be called after a variety of string related operations, these values make sense. We will return to this notion of *asymmetric confidence* later in this chapter.

The measure *OneAddedConfidence* represents how often a new instance of a pattern is created due to adding exactly one function call to the source code. This is calculated by dividing the number of times a new instance of this pattern is added to the source by adding exactly one function call to the code by the number of times a new instance of this pattern is added to the source by any means. The inverse of this values, T*woAddedConfidence* (*1 - OneAddedConfidence*), measures how often new instances of a pattern are created by adding both function calls to the source code at

once. A high value for two-added confidence signals a pattern that is very often correctly inserted by developers. This may mean that the developers using this pattern understand it very well. A high value for one-added confidence may signal an extremely buggy pattern, one that developers often need two source code changes to insert correctly or some large scale refactoring. If the two functions called in a usage pattern are created in the code at different times it is likely that the function usage pattern will emerge by adding a call to the newer function near a call to the older function. This will cause the *OneAddedConfidence* to be high until sufficient numbers of new instances of the pattern are created by adding both functions.

Finally, we keep track of a simple *Count* of how many times an instance of a function usage pattern is added to the source code. Higher values of *Count* (along with a high value of *DataFlowConfidence*) may indicate that a function usage pattern is more likely to be valid. Table 10 contains a list of the statistics computed for each function usage pattern and a brief description.

Metric	Description
DataFlowConfidence	Percent of the time a data flow relationship exists for this pattern
ConfFirst	How often, when the first function is added to the code, is a new instance of the pattern created
ConfSecond	How often, when the second function is added to the code, is a new instance of the pattern created
OneAddedConfidence	How often, when exactly one function is added to the code, is a new instance of the pattern created
Count	Raw count of new instances of the pattern

Table 10: Function Usage Pattern Statistics

5.3 Case Studies

We have applied each of our tools in this chapter in three case studies. The Apache httpd and Wine projects, discussed in sections 4.3.2 and 4.3.5 respectively, were both used as test cases. Additionally, student projects gathered at the University of Maryland were used. The Apache httpd project and the Wine project were both introduced in Chapter 4. Here we introduce the student data.

The student projects we analyzed were from an introductory C class at the University of Maryland. The data was collected by the Marmoset project [65]. The class consisted of 6 programming assignments, 5 of which we analyzed. The other assignment consisted primarily of the students writing test cases for their code. These test cases intentionally misused the APIs to test robustness as well as using the API correctly to test for correct behavior. Since students were deliberately leaving "bugs" in the code, our tool would be unable to identify the correct usage of the API. For each assignment, we have snapshots from each time the student submits the source code to an automated testing server called Marmoset [65]. Submissions are made to submit a project for a final grade and to see the results of an automated test suite provided by the instructors.

We analyzed 261 projects (we call one student's solution for one assignment a project), with a total of 4065 submissions. The projects had between 1 and 120 submissions, and averaged 15 submissions. Twenty-five projects had three or fewer submissions, 31 projects had 30 or more submissions. For each project, we mined each revision to build up a set of function usage patterns specific to that student's

code. These function usage patterns were then used to search the final submission of each project for violations.

The advantages of analyzing student projects are that they are small, making them quick to analyze, and that they are easy to read and understand (when analyzing reports of violations). This does, however, cause some concern for our work. Projects with a small number of submissions may have huge changes between successive submissions, thereby hiding many of the small bug fixes that were applied between submissions. The small size of the project may also lead to very few instances of any one particular function usage pattern.

5.4 Bug Finding

We have implemented a tool that will, given a set of function usage patterns, identify instances in the source code where one of the patterns is violated. These are instances where one of the function calls involved in the pattern exists in a function and there is no path through the function that contains the other function call in the correct position. For each violation a warning is produced listing the line number of the existing function call. As with the work discussed in Chapter 4, this work is purely based on intraprocedural analysis.

Even with the addition of our data flow analysis component, our tool discovers a large number of function usage patterns. In order to not overwhelm the developer with false positives we must carefully filter the function usage patterns we use to search for violations. We do this filtering based on the statistics we generate for each pattern. The filtering criteria have been experimentally derived by examining large numbers of violations produced by lightly filtered patterns and focusing on the ranges

that give the best false positive rate. We describe the statistics we used to filter the warnings produced by our tool in section 5.2.2. The values used to filter the function usage patterns to identify violations are listed in Table 11.

Parameter	Value
Count	>= 3
DataFlowConfidence	>=0.80
Number of Violations	<= 4

Table 11: Threshold Values, Bug Finding

In selecting function usage patterns to use to search for violations, we apply a number of criteria. First, we select patterns that have a *DataFlowConfidence* of at least 0.8. That means that at least 80% of the time when that pattern is added to the source code, it involves some type of data flow relationship. We also look at the C*ount*, the raw number of times a pattern is added to the code. Once a pattern has been added to the code three or more times, it is used to search for violations.

A violation of a function usage pattern can manifest itself in two ways: either the first function call can be missing or the second function call can be missing. Because of the notion of *asymmetric confidence*, a different confidence measure (*ConfFirst* or *ConfSecond*) is used depending on which function in the pattern is missing. If the second function in the pattern is missing, *ConfFirst* is used as the confidence measure. This metric is used because *ConfFirst* measures how often, when the first function in the pattern (the function call that is present in this type of violation), the pattern is completed. Likewise, *ConfSecond* is used when the first function call in a pattern is missing.

After the warnings are produced, they are filtered and sorted. They are filtered by their frequency; any function usage pattern that produces more than four warnings has all of its warnings discarded. A metric *PercentBroken* is also computed. This metric

is the number of times a pattern is broken in the source code divided by the number of times the pattern exists in the latest version of the source code. Any pattern that produces more warnings than correct usages in the source code is discarded. Note that any pattern that is not currently used in the latest version of the source code is always discarded by this filter. This combats the fact that some function usage patterns become invalid and disappear from the code. Finally, we sort the warnings by *OneAddedConfidence*, the confidence measure for the pattern (either *ConfFirst* or *ConfSecond*) and then by *PercentBroken*. The use of *OneAddedConfidence* is key. This value represents how often a new instance of a particular pattern was created in the code by adding exactly one function call. *OneAddedConfidence* is a metric that cannot be computed from merely looking at the latest version of the source code, and the fact that we have found it to be a good predictor of bugs indicates that patterns that have a history of being broken in the past continue to be buggy.

5.4.1 Results

We applied our bug finding tool in three case studies. The Apache httpd and Wine projects were each used as a case study. Additionally, student projects gathered at the University of Maryland were used. The warnings that are deemed to be likely bugs in this section are declared so for many reasons. If the flagged code could cause incorrect behavior, including a crash, the warning was declared a likely bug. Additionally, broken function usage patterns that are likely to cause software maintenance issues in the future are also deemed to be likely bugs. An example of the latter case arises when the internal fields of a data structure are manipulated rather than going through the defined accessor functions.

```
addr = atoi((line+(*j)));
if(addr>-1 && addr<65536)
    return(addr);
else
    return(-1);
(*j) += numDigits(addr);
```

Figure 17: Student Project Bug, Function Usage Pattern partially in Dead Code

Student Projects

In the final projects submitted by the students, our static analysis tool found 36 violations of the function usage patterns that we deemed to be worth looking at based on the statistics generated for the violated function usage patterns. Of these, we believe 12 are actual bugs. These 36 warnings were spread out over only seven individual projects. An example bug is shown in Figure 17. In this example, the student has written the entire function usage pattern correctly (`atoi -> numDigits`), however the function call to `numDigits` resides in dead code.

A common source of bugs in the student code was the result of a student rewriting a function inline (sometimes incorrectly) instead of calling a function that had already been written. This often happened with accessor functions and helper functions. This type of bug was also seen in the open source projects (see below). Note that it is necessary for the code to sometimes use these accessor or helper functions correctly in order for the tool to learn the pattern. In many of these cases, the developer has created a data structure and provided some functions to access that structure. These functions are often provided to allow the underlying data structure to be changed without causing every access to the data structure to be updated. Instead, only the accessor functions need to be updated. While this type of bug was found in the open

source projects, it is especially worrisome in the student code. Part of the motivation for these coding projects is to teach students how to create and use abstract data structures.

Apache httpd

Our tool found a large number of violations of the function usage patterns in the Apache httpd source code and ranked them as specified. We examined the top 51 warnings. Note that all violations missing the same function call of the same function usage pattern are ranked the same. The 50th and 51st warnings were both produced from the same function usage pattern so there was no way to choose one to examine and one not to examine. We identified 11 likely bugs (see Appendix A for the full 51 warnings and the likely bugs). The main sources of false positives are the use of wrapper functions, irregular API usage and replacing a function call that generated data by passing that data as an argument to the function that contains the warning. The sources of false positives are discussed in section 5.4.2.

```
apr_thread_mutex_t * apr_allocator_mutex_get(
                                    apr_allocator_t *allocator);

. . . .
apr_allocator_t *allocator;

. . . .

if (allocator->mutex)
      apr_thread_mutex_lock(allocator->mutex);
```

Figure 18: Apache httpd bug, Access data structure internals

An example bug is show in Figure 18. This code snippet shows an error caused by not using the correct accessor function to manipulate a data structure. Listed first is the prototype of the function used to provide access to a mutex stored in the data type,

the `struct apr_allocator_t`. Following that are an instance of a pointer to the data type and the buggy snippet of code. Rather than accessing the mutex field directly, both accesses should go through the accessor function, `apr_allocator_mutex_get`. This bug will not cause any immediate problems; the code will work correctly as written. However, changes to the internal representation of this struct may break this code. This coding practice may cause problems in the future and creates code that is difficult to maintain. By providing an accessor function as part of this data type's API, the developers of the data type have protected the user of the data type against any internal changes that might take place in the implemetation of the struct. In this case, if the field `mutex` in the struct is renamed the code in Figure 18 will need to be rewritten. Three instances of this particular bug were found in the Apache httpd source code.

The evaluation that needs to be made is to determine whether or not the data mined from the revision history adds value to the tool. Could we collect data from only the last version of the source code and receive the same benefit? To investigate this, we sorted the warnings our tool produced by *PercentBroken* (in ascending order). This metric compares how many times a function usage pattern is used correctly against how many times it produces a warning. Each of these values are calculated by looking only at the latest version of the source code. The violations of function usage patterns that are often used correctly in the source code and less often broken in the source code rise to the top. The top 51 warnings according to PercentBroken were inspected and three likely bugs were found. A Chi-square test was run in a similar fashion to the one described in section 4.4.3. The Chi-square value was 5.2987 and

the p value was less than or equal to 0.025, denoting that the difference was statistically significant. Table 12 shows the Chi-square calcuation.

	Ranking with Historical Information	Ranking without Historical Information	Total
Likely bugs	11	3	14
False Positives	40	48	88
Total	51	51	102

Table 12: Chi-square Calculation, Apache httpd Pattern Violations

Wine

We applied our tool to the Wine source code to identify likely bugs. This found a large number of violations of the function usage patterns and ranked them as specified. Emulating how a developer would use a static analysis tool, we examined the top 51 warnings. Twenty-seven likely bugs were identified. See Appendix B for the full 51 warnings and the likely bugs. The main sources of false positives are the use of wrapper functions, irregular API usage and replacing a function call that generated data by passing that data as argument to the function that contains the warning. The sources of false positives are discussed in depth in section 5.4.2

Figure 19 displays a bug from the Wine source code that our system was able to detect. In this example, we have correctly inferred that a call to `socket` needs to be followed by a call to `close`. The call to `close` is missing before the return statement is reached. Since `fd` is a local variable, and not returned out of the function, it could not be closed except in this function. This particular bug has been fixed in the recent Wine source code by adding a call to `close` before line 7[3]. This

[3] We used a CVS snapshot from 14 September 2004 for our mining and then used recent history to validate these bug fixes.

function usage pattern was violated four times in the source code. Three of the violations are likely bugs (two of those violations were later fixed and the author submitted the third as a bug report and received a positive response) and one is not a bug since the descriptor escapes the scope of the function in global data. This set of warnings gives a good illustration of how this system works and how it would benefit a development team to mine each revision as it is made. These four warnings dealing with a missing call to `close` are not ranked very high in the list of warnings produced by the tool. They are ranked down about the 300th warning. However, as each of these bugs is fixed, the remaining bugs will begin to rise to the top of the list.

```
1  int fd = socket(PF_INET, SOCK_DGRAM, 0);
2  if (fd != -1)
3      if (ioctl(fd, SIOCGIFFLAGS, &ifr))
4          ret = ERROR_INVALID_DATA;
5       else
6          ret = NO_ERROR;
7  else
8      ret = ERROR_NO_MORE_FILES;
9  return ret;
```

Figure 19: Bug due to missing close call from the Wine Source Code

```
1 if (infoPtr->dwStyle & CCS_NORESIZE) {
2     uPosFlags |= (SWP_NOMOVE);
3     cx = 0;
4     cy = 0;
5     TOOLBAR_CalcToolbar (hwnd);
6  }
   . . .
7  return 0;
8
```

Figure 20: Bug due to missing InvalidateRect call from the Wine Source Code

Figure 20 displays another bug found by our tool in the Wine source code. In this case, the call to `TOOLBAR_CalcToolbar` needs to be followed by a call to `InvalidateRect` to ensure the new toolbar is drawn on screen correctly. This bug has also been fixed in a subsequent version of Wine by adding a call to `InvalidateRect` after line 5. This function usage pattern was violated three

times in the source code. Two of the violations are likely bugs (one of those violations being the instance that was later fixed) and one is not likely to be a bug since the missing function eventually gets called later in the call tree. The source code in these examples has been edited slightly for clarity.

As with the Apache httpd results, these results were evaluated against results produced by ranking the warnings without the use of historical information (ranking the warnings by *PercentBroken* only). The top 51 warnings were inspected and zero likely bugs were found. A Chi-square test was run in a similar fashion to the one described in section 4.4.3. The Chi-square value was 36.1288 and the p value was less than or equal to 0.001, denoting that the difference was statistically significant.

5.4.2 Sources of False Positives

The false positive warnings produced by our tool stem from three main problems. In general, while there are rules that govern how to use a set of functions, those rules do not necessarily need to be enforced within the bounds of one function. Our analysis is completely intra-procedural and it is often the case that a false positive was generated for the pattern `foo->bar` because the call to one of the functions in the pair was one function away in the call graph. Adding one level of interprocedural analysis, tracking what functions are called by the functions called by the current function, would have suppressed a large number of false positives. Related to this problem are wrapper functions and functions that implement part of a conceptual task. These are more prevalent in the open source projects than in the student code but caused problems in both. For example, a module may wrap a call to a general mutex function to provide access to a particular mutex within the model. These

wrappers call one half of a function pair (lock or unlock), and hence produce many false positives. It is also common in the Wine source code where wrappers are used to provide access to 16- and 32-bit library operations.

Another source of false positives are function usage patterns of the form *foo* -> *{bar_1, bar_2, ... bar_n}*, where the function *foo* needs to be followed by one function call from a set of functions. The false positives occur if the function *foo* is equally likely to be followed by any of these functions. If that is the case, when a function call to *foo* is not followed by *each* of these functions a warning will be emitted. An example of this comes from the Apache source code. The function call `ap_run_quick_handler` is used to attempt to quickly serve a request back to the connection if possible. If this fails, the request needs to be handled in a slower, but more reliable way. To do this, one of two function calls can be used: `ap_process_request_internal` or `ap_invoke_handler`. Only one of these functions needs to be called, however, depending on the situation. Since they are both used in these situations a warning will be produced to recommend that the other (uncalled function) is called. Another example of this from the Apache code involves the functions that are used to allocate strings. There are a number of functions that can be used to do this, but two are especially interesting. The functions `apr_pvsprintf` and `apr_psprintf` each take a variable number of arguments (a format string and set of values) and return a newly allocated string that contains the results of applying the values to the format string. These functions are functionally identical to `sprintf`, except that these functions allocate memory. In fact, the entire body of `apr_psprintf` is code that calls `apr_pvsprintf` with the

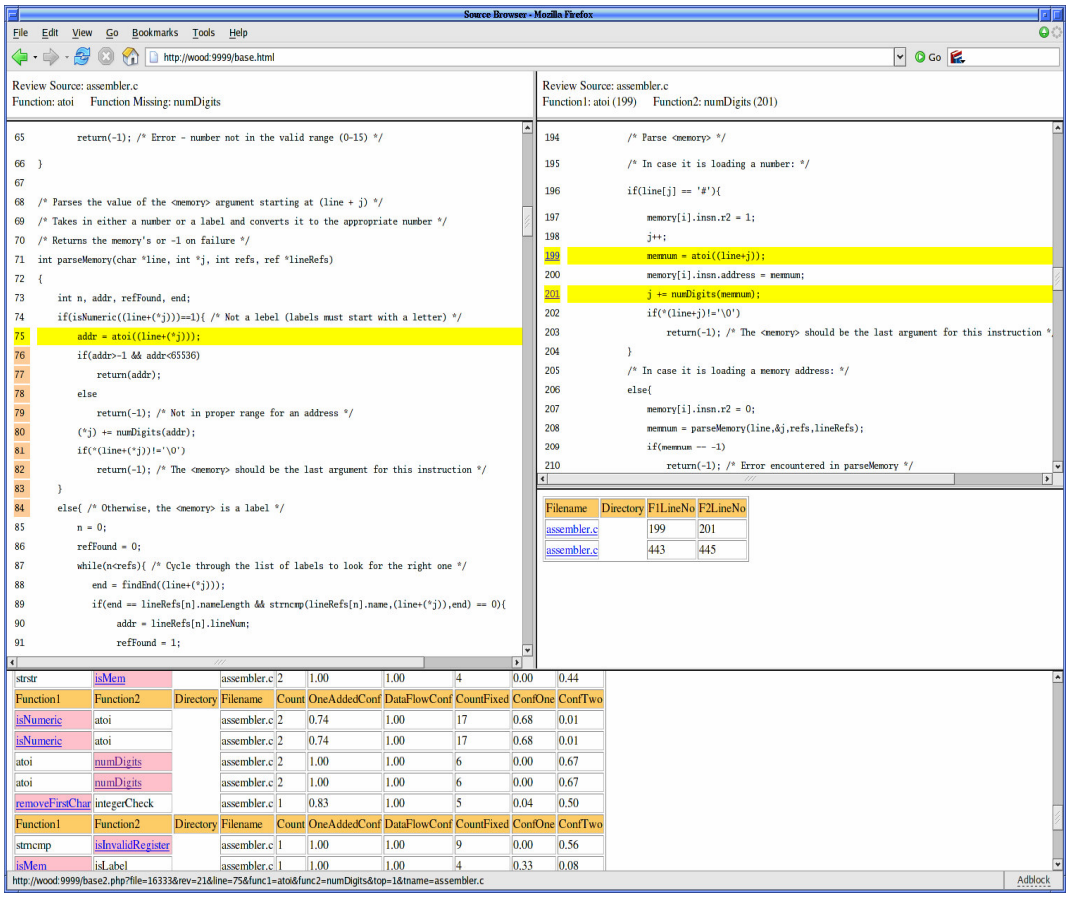
arguments unchanged. This allows the developer to choose which function to call to produce a new string from a format string and some values. The function `apr_psprintf` is called just over 2,000 times in the source code. The function `apr_pvsprintf` is called just over 500 times in the source code. This type of function duplication causes problems when mining out function usage patterns. To remove these false positives, the tool would need to conflate the two functions and ensure that either one of them gets called in the right place.

Another source of false positives is instances when there is a producer/consumer data flow relationship. These are cases where the first function called often produces data used by the second function call. False positives arise when the value produced by the first function call is passed into the calling function as a parameter or pulled out of an existing global data structure. This negates the need for calling this function and the pattern appears violated. Performing a more sophisticated data flow analysis could prune these false positives.

A source of false positives that so far appears to be unique to the Wine source code is the creation of stub functions to act as placeholders until the developers implement the function. The unique aspect of this is that the stub functions in Wine are often not empty. They contain a number of function calls that do not seem to achieve anything useful. Examples of this can be found in the `MCIAVI_mciXXX` functions. A number of these functions (22) are stub functions and all have the exact same implementation, a call to `FIXME` (a debug logging macro) followed by a call to `MCIAVI_mciStop` and `MCIAVI_mciGetOpenDev`. This code produces an invalid function usage pattern for the functions `MCIAVI_mciStop` and

`MCIAVI_mciGetOpenDev`, which are fully implemented and otherwise useful functions.

Finally, spurious function usage patterns are detected when a temporary variable in a function is reused. It is sometimes the case that an integer will be declared to catch all the return values produced by function calls in the body of a function. Reuse of this variable will cause data flow relationships to build up between neighboring function calls that may or may not actually form an interesting pattern.



The left pane displays the buggy code while the right pane displays an example of the function usage pattern applied correctly. The bottom pane displays the entire list of warnings, with the missing function call highlighted.

Figure 21: Warning Browser

5.4.3 Warning Browser

Along with our static analysis tool, we have built a web-based interface to present the warnings produced by our tool to the developer. Figure 21 shows the browser displaying a warning we determined to be a true bug in one of the student projects. The bug in this case involves the pattern `atoi -> numDigits`. This is an interesting instance of a bug in that both function calls exist in the code, but the call to `numDigits` (on line 80) is in a region of dead code in the function (this is the same example in Figure 17). The bottom pane of the GUI lists the warnings produced by our tool along with the statistics mined out of the change history describing the violated function usage pattern. The missing function is highlighted and acts as a link to display the source code for the violation in the top left pane. In this pane, the existing piece of the function usage pattern is highlighted and the line numbers before or after it are shaded to let the developer know whether the missing function call is expected to be added before or after the existing function call. The two upper panes on the right side of the GUI display examples of the function usage pattern in use in the current version of the source code. The lower pane lists a number of locations in the code where the pattern is correctly applied and allows the developer to select one of the examples to display; the upper pane displays the selected source code. While useful to us, the bottom pane could be improved for general use.

5.5 Documenting API Usage

One of the examples that motivated this work was the idea of recovering the rules that define how to use the functions defined in a particular API. While finding bugs

based on these patterns is useful to the software project, identifying high probability rules that describe an API, and have not yet necessarily caused bugs, could serve to *prevent* bugs by documenting to the developer rules to follow. We have shown how to identify meaningful function usage patterns and gain some insight into how troublesome these patterns have been in the past, but we still need to be able to discover a set of rules that defines how to use an API. This section presents our tool that automatically documents an API but identifying function usage patterns relevant to the functions in that API.

Unfortunately, the C programming language does not provide a specific way to denote that an API consists of a particular set of functions. With C++ or Java, an API is implicitly defined to be the set of methods exported by an object. Java also allows the developer to specify an API using the interface construct. The C language does offer the header file as a place to define an API, but this is just a suggestion. We inspected APIs defined by header files but found these to be incomplete. There are many cases where developers place functions in a C file and use the `extern` construct, rather than a prototype in a header file, to make these functions visible in other C files.

There are a number of heuristics that may be used to denote a set of functions as part of an API. These include looking for a common prefix or suffix in a function name, grouping functions by common arguments and defining the functions in a common location (either in the same file or same directory). In our work, we use heuristics based on the location of the function definition. The first heuristic is to consider the set of functions defined within the same source file as part of the same

API. The second heuristic we consider is to assume functions defined within the same directory in the code tree are part of the same API. From inspecting the source code in our case studies, these two heuristics seem to be the most widely used.

The API documentation is built by looking at what function usage patterns involve two functions defined in the API. The function usage patterns we ascribe to the API is filtered to contain those patterns that have a high level of support, in particular those with a *DataFlowConfidence* of at least 0.8.

These function usage patterns give a partial view of how to interface with an API. The interaction with an API is not limited to a sequence of two function calls. More likely, a series of function calls needs to be made. It is also likely that at some point in the chain of function calls, calling any function call from a set of function calls is possible. This is because of the *asymmetric confidence* of some function usage patterns. Take for example, a data structure that is protected by a mutex. It is very likely the case that the function call chain may look like Figure 22. `Lock` must be followed by one of the functions listed in the brackets. In this example, the function usage pattern `Lock->getFoo` exists. However, the confidence metrics of this relationship are asymmetric, `getFoo` always need to be preceded by `Lock`, but `Lock` does not always need to be succeeded by `getFoo`.

```
Lock(); [getFoo()|getBar()|updateFoo()|updateBar()] Unlock();
```

Figure 22: Call Chain with Variable Middle Function Call

To determine these longer chains of function calls, we mine the function usage patterns for patterns that fit the form described in Figure 23. With this, function usage patterns of length three can be recovered. This technique could be used to

produce function call chains of arbitrary length, as long as enough high confidence patterns exist. However we will restrict the scope of our work to function call chains of length three.

```
Foo() -> Bar()  Bar() -> Zoo()     Foo() -> Zoo()
```

Figure 23: Function Usage Patterns for Chain Creation

To evaluate this work, we will examine how many of the function usage patterns discovered involving only functions defined in an API are correct. Unfortunately, the APIs within the open source projects we have studied are not well documented, making it difficult to validate the mined function usage patterns. This lack of documentation motivates this work but also makes it difficult for a non-expert to determine correctness. In order to evaluate the performance of this tool, we will examine the function usage patterns describing external, well documented APIs used by the source code. For the Apache httpd project these APIs include the various APIs included with the SSL library. For the Wine project, the socket library will be evaluated. While the Wine project does provide, publicly, an implementation of the Windows API, internally, little of this API is actually used. For the student data, the API for the C library memory functions, including `malloc` and `free`, will be examined. Each of these APIs is well understood and the challenge for the tool is to be able to discover function usage patterns that correctly document usage of the API. It may not be possible, from examining one software project, to produce documentation for the full API. Only the parts of the API used by the source code and that form intraprocedural patterns are documented.

5.5.1 Results

Student Projects

We mined the student data for rules that represent an internally defined API but were unsuccessful in producing a full API out of any of the student projects. We believe there are a number of reasons for this. Each of the student projects were small in terms of depth of the CVS history and in the size of the code base. Most of the APIs developed by the students provide access to an abstract data type. These APIs are small in terms of the number of functions involved. They are also only used once or twice in each student project, providing a small number of new instances of function usage patterns. Invocations of initialization and finalization functions are especially rare. Finally, each student project was contained in one directory with no subdirectories. As a result of this, the APIs were mined from functions defined in the same file, which may or may not represent and entire API.

The student projects were also mined for function usage patterns that describe the memory management functions defined in libc. These functions include `malloc`, `calloc` and `free`. The `mmap` family of functions is also included in this API but the student code makes no use of them. The tool identified five separate function usage patterns defined between functions in this API. Each were valid patterns. The two key patterns for this API, `malloc` followed by `free` and `calloc` followed by `free`, were both found. Two other patterns, `malloc` called after `malloc` and `calloc` followed by `calloc`, are useful as well. These latter two patterns denote instances where a struct is allocated and then individual fields (often arrays) are allocated using the same function call. The final function usage pattern found for this

API was `malloc` followed by `calloc`. This pattern was discovered in 5 separate projects, each submitted by a different student. In each case, the student used `malloc` to allocate a struct of some type and `calloc` to allocate memory for individual pointers within the struct. For these cases, the function `calloc` appears to be used to allocate memory for strings or for arrays. This appears to be a more sophisticated use of the API. Rather than use `malloc` and `memset` to zero out a data structure the students were able to recognize the correct usage of `calloc`.

Apache httpd

We use the subdirectories in the Apache httpd source code to build a set of rules to explain how to use a particular API. A large part of the Apache source code is code to implement loadable modules to add functionality to the web server. Each module has its source code in one directory; however some directories contain the source code for more than one module. This makes looking for an API on a directory boundary rather than at a file boundary more appropriate. We were able to reconstruct a reasonable set of API rules for each directory. One challenge this presents is that an API may have one set of functions meant for use by the public and another set of functions for internal use. Again this is a place where the C programming language does not make discovering a public interface easy. Since we are studying the source code for an API, rather than source code that uses an API, we may have trouble identifying the set of rules that explain how a *user* is to interact with the public functions in the API.

Apache provides a compatibility library to insulate its code from variations in the various platforms it has been ported to [4]. Part of this library is code to provide

access to mutex functionality. On Linux, the platform used for this work, the locking mechanism is implemented on top of the pthreads library. The library code wraps the pthread calls and also adds some functionality specific to Apache. The function usage patterns mined from this directory (srclib/apr/locks/unix) are shown in Table 13. In this table it is interesting to note that there are three levels of locking provided by the library, a global lock, a process lock and a thread lock. Each level has its own set of functions which form a smaller API within the directory. The locks provided at the thread level are the most complex, providing not only vanilla mutex functionality but also read and write locks (providing *rdlock* and *wrlock* functions, respectively). It is also interesting to note that the functions that provide the functionality to destroy the mutex are not present in Table 13. This appears to be caused by the fact that the destroy functions are usually called in clean up functions that only operate on one level (global, process or thread) of data. Therefore, they are segregated out from the other locking functions and from each other. They do form function usage patterns with other functions in the Apache code, just not functions defined in this API.

```
apr_thread_rwlock_rdlock(rwmutex);
x = guardedData.value;
apr_thread_rwlock_unlock(rwmutex);

. . .

apr_thread_rwlock_rdlock(rwmutex);
y = guardedData.value;
apr_thread_rwlock_unlock(rwmutex);
```

Figure 24: Lock Sequence

Of the patterns in Table 13, most are valid usage patterns. The patterns that are not necessarily valid are those that are of the form `XXX_unlock -> XXX_xxlock`. These are due to lock and unlock functions being called in a series as shown in Figure

24. Here a particular data structure is being protected by a mutex and each access in a function needs to be guarded.

Function One	Function Two	Count	ConfFirst	ConfSecond
apr_global_mutex_child_init	apr_global_mutex_child_init	2	0.20	0.40
apr_global_mutex_lock	apr_global_mutex_unlock	6	0.50	0.50
apr_proc_mutex_lock	apr_proc_mutex_unlock	6	0.75	0.67
apr_thread_cond_wait	apr_thread_mutex_unlock	3	0.50	0.04
apr_thread_mutex_create	apr_thread_cond_create	2	0.50	0.33
apr_thread_mutex_create	apr_thread_mutex_lock	3	0.13	0.02
apr_thread_mutex_lock	apr_thread_cond_wait	3	0.08	0.50
apr_thread_mutex_lock	apr_thread_mutex_unlock	128	0.88	0.65
apr_thread_rwlock_create	apr_thread_rwlock_rdlock	3	1.00	0.38
apr_thread_rwlock_create	apr_thread_rwlock_unlock	3	1.00	0.18
apr_thread_rwlock_create	apr_thread_rwlock_wrlock	3	1.00	0.30
apr_thread_rwlock_rdlock	apr_thread_rwlock_unlock	9	0.69	0.41
apr_thread_rwlock_rdlock	apr_thread_rwlock_wrlock	2	0.25	0.20
apr_thread_rwlock_unlock	apr_thread_rwlock_rdlock	4	0.24	0.50
apr_thread_rwlock_unlock	apr_thread_rwlock_unlock	6	0.35	0.35
apr_thread_rwlock_unlock	apr_thread_rwlock_wrlock	2	0.12	0.20
apr_thread_rwlock_wrlock	apr_thread_rwlock_rdlock	3	0.30	0.38
apr_thread_rwlock_wrlock	apr_thread_rwlock_unlock	10	0.67	0.45
apr_thread_rwlock_wrlock	apr_thread_rwlock_wrlock	2	0.20	0.20

Table 13: Function Usage Patterns, APR Locks, Apache

Building an API for the SSL libraries [55], an external API for a third party library, 43 function usage patterns are produced. See

Table 14: SSL Library API, Recovered from Apache httpd source code for a list of function usage patterns. These function usage patterns are restricted to functions defined in the SSL library. The Apache httpd source code that accesses these functions is contained in the code for `mod_ssl`, the loadable module used to add SSL support to the web server. Each of these patterns was validated by hand against the documentation. Most of these function usage patterns have an *OneAddedConf* of zero, meaning instances of the pattern are most often added correctly.

The SSL library does not provide a large array of functionality. There are three main phases of the use of the library: create a connection, use a connection and

shutdown a connection. The vast majority of function usage pattern describe creating a connection, the most complicated part by far.

We do see a distinction between the `SSL_*` functions and the `SSL_CTX_*` functions. The former are used for dealing with individual connections and the latter are used for dealing with the context data structure (`SSL_CTX`) which holds default value for creating individual SSL connections.

Ten of these patterns involve functions that allow access to application specific data stored in the SSL data structures. Since these function call deal with application specific data, their complete usage patterns may depend upon the application that is using the SSL library. However, the function usage patterns mined from the repository that deal with these functions show that these functions are generally used during initialization of a connection or when data is read from a connection. While application specific data is optional, storing it into the SSL data structure when a connection is created appears to be a logical use. Further, the application specific data stored by the Apache httpd code is used to produce application and connection specific error messages in response to SSL errors. This allows the Apache httpd code to pass around only SSL data structures and maintain some connection specific state. This appears to be a key design feature of the SSL library used correctly by the Apache httpd source code and correctly identified by the function usage patterns.

Function1	Function2	Count	ConfOne	ConfTwo
SSL_accept	SSL_get_error	3	0.50	0.13
SSL_accept	SSL_get_verify_result	2	0.50	0.40
SSL_accept	SSL_get_peer_certificate	2	0.50	0.20
SSL_CTX_ctrl	SSL_CTX_set_ex_data	3	0.13	0.50
SSL_CTX_load_verify_locations	SSL_CTX_get_client_CA_list	3	0.50	0.50
SSL_CTX_new	SSL_CTX_set_ex_data	3	0.33	0.50
SSL_CTX_set_ex_data	SSL_CTX_ctrl	3	0.50	0.13
SSL_CTX_set_tmp_rsa_callback	SSL_CTX_set_tmp_dh_callback	2	0.67	0.67
SSL_CTX_set_verify	SSL_CTX_get_client_CA_list	3	0.75	0.50
SSL_CTX_set_verify	SSL_CTX_load_verify_locations	3	0.75	0.50
SSL_do_handshake	SSL_state	4	0.50	0.33
SSL_get_ex_data	SSL_accept	2	0.13	0.50
SSL_get_ex_data	SSL_connect	2	0.11	0.50
SSL_get_peer_certificate	SSL_get_verify_depth	2	0.13	0.50
SSL_get_SSL_CTX	SSL_get_peer_certificate	2	0.50	0.13
SSL_get_verify_result	SSL_get_peer_certificate	4	0.57	0.25
SSL_new	SSL_set_tmp_rsa_callback	4	0.50	0.50
SSL_new	SSL_set_tmp_dh_callback	4	0.50	0.50
SSL_new	SSL_set_ex_data	4	0.50	0.50
SSL_new	SSL_set_session_id_context	4	0.50	0.40
SSL_new	SSL_set_verify_result	4	0.50	0.33
SSL_read	SSL_get_error	3	0.60	0.13
SSL_read	SSL_get_ex_data	2	0.67	0.07
SSL_renegotiate	SSL_do_handshake	3	0.50	0.13
SSL_renegotiate	SSL_state	3	0.50	0.08
SSL_set_cipher_list	SSL_get_peer_certificate	2	0.50	0.13
SSL_set_ex_data	SSL_set_tmp_rsa_callback	4	0.50	0.50
SSL_set_ex_data	SSL_set_tmp_dh_callback	4	0.50	0.50
SSL_set_ex_data	SSL_set_verify_result	4	0.50	0.33
SSL_set_session_id_context	SSL_set_tmp_rsa_callback	4	0.40	0.50
SSL_set_session_id_context	SSL_set_tmp_dh_callback	4	0.40	0.50
SSL_set_session_id_context	SSL_set_ex_data	4	0.40	0.50
SSL_set_shutdown	SSL_free	6	0.55	0.55
SSL_set_tmp_dh_callback	SSL_set_verify_result	4	0.50	0.33
SSL_set_tmp_rsa_callback	SSL_set_tmp_dh_callback	4	0.50	0.50
SSL_set_tmp_rsa_callback	SSL_set_verify_result	4	0.50	0.33
SSL_state	SSL_accept	3	0.23	0.50
SSL_state	SSL_connect	2	0.17	0.50
SSL_write	SSL_get_error	2	1.00	0.10
SSLv2_client_method	SSL_CTX_new	2	0.50	0.33
SSLv2_server_method	SSL_CTX_new	4	0.50	0.33
SSLv23_client_method	SSL_CTX_new	2	0.50	0.33
SSLv23_server_method	SSL_CTX_new	5	0.50	0.36

Table 14: SSL Library API, Recovered from Apache httpd source code

The list of function usage patterns in Table 14 does demonstrate the basic patterns for various pieces of a program that needs to use SSL. The setup phase of SSL requires that the applications define whether it will act as a server or client and which version of SSL that will be used. This is documented in the `SSLv##_XXX_method` calls being required before `SSL_CTX_new`. These patterns do accurately reflect how the library needs to be used. Further, the patterns that start with a call to the function `SSL_new` document what needs to be done to create a new SSL connection. In the table above, `SSL_new` is followed by functions to setup callbacks, store application specific data with the connection (`SSL_set_ex_data`) and set up the session identifier. These patterns are accurate reflections of how to use the library. From investigating the documentation provided for the SSL library [55], the only other functions from the API that are likely to be called in close proximity to SSL_new are `SSL_set_bio` and `SSL_set_fd`. In the Apache httpd code `SSL_set_fd` is never used. The function `SSL_set_bio` is called in one location in the code that is separated from the `SSL_new` invocation. In summary, the function usage patterns that are found for `SSL_new` are correct, but not complete.

Both `SSL_read` and `SSL_write` form patterns with the function `SSL_get_error` which is used to retrieve any error codes those functions produce. The `SSL_read` function is also paired with `SSL_get_ex_data`. This function returns application specific stored with the connection. Its use by Apache was discussed previously. While every SSL application may not exercise this pattern, it reflects the correct way to use this part of the API.

Overall, the function usage patterns listed in Table 14 are correct. However, the documentation for the API is not complete. This is due to both the fact that Apache httpd does not exercise the entire API and the structure of the Apache httpd code prevents some patterns from being found by an intraprocedural analysis. The tool was not able to document any of the functions from the SSL library involved with the `SSL_SESSION` data structure. This data structure is used to contain details about a specific SSL session. It appears that the Apache code referencing these functions is very small. Each function is referenced only once or twice, and that this part of the Apache code has not been updated. Since there are very few new instances of the function usage patterns involving these functions, our tools do not yet recognize them as valid. The same is true for the `SSL_alert_*` functions, used to signal the machine on the other end of the connection of a special situation. This is a special message sent over the connection that is separate from the normal data stream. The alert functions that are used (two functions of a possible four defined by the library) are each invoked once and that snippet of code has not been updated since first being written.

Wine

We use the subdirectories in the Wine source code to try to build a set of rules to explain how to use a particular API. A large part of the Wine source code is code to implement dynamically linked libraries, and the source code is organized into one directory per library. This makes looking for an API on a directory boundary rather than at a file boundary more appropriate. We did this analysis at the file boundary as well, but because of the structure of the Wine code this was less effective. While we

did highlight some interesting patterns in the Wine source code by doing this, the structure of the source code makes the smallest logical module the directory. A review of these file-defined sets reveals an incomplete description of an API. We were able to reconstruct a reasonable set of API rules for each directory. One challenge this presents is that an API may have one set of functions meant for use by the public and another set of functions for internal use. Again this is a place where the C programming language does not make defining a public interface easy. Since we are studying the source code for an API, rather than source code that uses an API, we may have trouble identifying the set of rules that explain how a *user* is to interact with the public functions in the API. We did, however, find some interesting and unexpected results.

In the Wine source code, the subdirectory wine/dlls/msi contains code to provide access to a database and perform queries. Table 15 contains a selection of function usage patterns defined on functions implemented in the wine/dlls/msi directory and selected statistics. Each function usage pattern listed in this table has a *DataFlowConfidence* of one. This data is sorted by Function1 and then by *Count*. The function defined in separate files, but in the same directory, all have confidence values less than the threshold value. Patterns dealing with specialized functions, for example MSI_`RecordGetInteger`, also have lower confidence values. This is because not every record will contain a specific value, an integer in this case, to retrieve.

The file names listed in the last two columns denote the files that contain the implementations of Function1 and Function2, respectively. Browsing through the

files in the directory one can see that there are two sets of functions with very similar names. There are the functions with a name of the form `MSI_Xxx` and functions with a name of the form `MsiXxx`. The rules produced for the functions defined in this directory are almost exclusively relationships between the `MSI_Xxx` class of functions. There are 35 such rules that have at least one instance in the current version of the software. There are only two rules defined between `MsiXxx` functions. In looking through the documentation, it appears that the `MsiXxx` functions are the publicly exposed functions for dealing with a database while the `MSI_Xxx` functions are merely for internal use.

Function1	Function2	Count	Conf-First	Conf-Second	Filename	Filename
MsiOpenDatabaseW	MsiCloseHandle	4	0.31	0.01	msi.c	handle.c
MSI_DatabaseOpenViewW	MSI_ViewClose	32	0.55	0.23	msiquery.c	msiquery.c
MSI_DatabaseOpenViewW	MSI_ViewExecute	32	0.55	0.48	msiquery.c	msiquery.c
MSI_ViewExecute	MSI_ViewClose	32	0.48	0.23	msiquery.c	msiquery.c
MSI_DatabaseOpenViewW	MSI_ViewFetch	29	0.50	0.48	msiquery.c	msiquery.c
MSI_ViewExecute	MSI_ViewFetch	29	0.44	0.48	msiquery.c	msiquery.c
MSI_ViewFetch	MSI_ViewClose	29	0.48	0.21	msiquery.c	msiquery.c
MSI_ViewFetch	MSI_RecordGetStringW	15	0.25	0.13	msiquery.c	record.c
MSI_ViewFetch	MSI_RecordIsNull	10	0.17	0.13	msiquery.c	record.c
MSI_ViewFetch	MSI_RecordGetInteger	7	0.12	0.18	msiquery.c	record.c
MSI_RecordIsNull	MSI_RecordGetStringW	26	0.33	0.23	record.c	record.c
MsiCreateRecord	MsiRecordSetStringW	22	0.35	0.28	record.c	record.c
MSI_CreateRecord	MSI_RecordSetStringW	10	0.43	0.30	record.c	record.c
MSI_RecordIsNull	MSI_RecordGetInteger	10	0.13	0.26	record.c	record.c

Table 15: Selected Function Usage Patterns from the wine/dlls/msi directory

In terms of documenting the internal API of the code we have discovered a number of interesting things. First, there are the rules themselves. Second, we see that when writing internal code, developers should not be mixing calls to the two sets of functions. Third, if we lift the requirement that the function usage patterns we report must have an instance in the current version of the software we find that the change

history suggests that the same relationships that we have reported among the `MSI_Xxx` did exist among the `MsiXxx` functions, but are no longer relevant in the code. By looking back through our mined data, we can see the instances of the patterns involving only `MsiXxx` functions disappeared on 9 July 2004. This commit changed nearly all calls of `MsiXxx` to calls of `MSI_Xxx`. The commit message for this change was "Refcount all objects, and use pointers internally."

The filenames listed in Table 15 suggest that looking on a file boundary would identify some, but not all, function usage patterns in the API. Four of the listed function usage patterns cross a file boundary. From the examples, it is clear that one source file contains code to run a query while another contains code to access records returned by the query.

Function One	Function Two	Function Three	Count
NdrProxyGetBuffer	NdrProxySendReceive	NdrProxyFreeBuffer	70
NdrProxyInitialize	NdrProxySendReceive	NdrProxyFreeBuffer	70
NdrProxyInitialize	NdrProxyGetBuffer	NdrProxyFreeBuffer	70
NdrProxyInitialize	NdrProxyGetBuffer	NdrProxySendReceive	70
MSI_DatabaseOpenViewW	MSI_ViewExecute	MSI_ViewFetch	29
new_res	put_res_header	set_dword	18
put_res_header	set_dword	put_pad	18
PSDRV_WriteGSave	PSDRV_WriteFill	PSDRV_WriteGRestore	7
PSDRV_SetClip	PSDRV_Brush	PSDRV_ResetClip	6
PSDRV_WriteGSave	PSDRV_WriteStroke	PSDRV_WriteGRestore	4
PSDRV_WriteGSave	PSDRV_Clip	PSDRV_WriteGRestore	4
PSDRV_WriteGSave	PSDRV_WriteHatch	PSDRV_WriteGRestore	4

Table 16: Function Usage Patterns, Chains, Wine Source Code

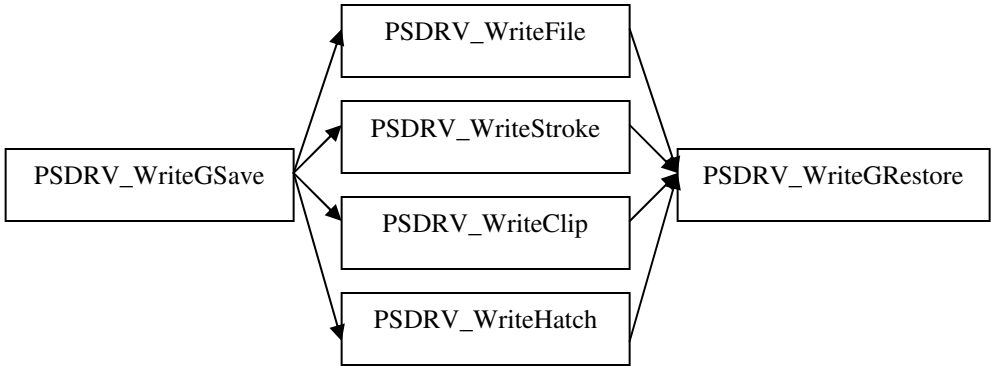


Figure 25: Graph of PSDRV_WriteXXX Functions

We also examined chains of length three of function usage patterns constructed per API. There were 200 chains of length three identified by our tool that appear in the latest version of the source code. Of these 200 chains, 75 appear in the source code more than twice.

Table 16 shows a selection of these chains from a number of APIs. The function calls in the chain are listed in order. The column labeled *Count* denotes how many times this chain was identified in the latest version of the source code. Looking at the first four chains listed, it is easy to see how larger and larger chains could be built. The chains that start with `PSDRV_WriteGSave` and end with `PSDRV_WriteGRestore`, demonstrate an example of a set of chains that represent a pattern where the middle function invoked can be one of any set of functions. A graph representing these relationships is shown in Figure 25. Arrows point to functions that need to be called after the function at the starting point of the arrow.

The socket API (functions defined in sys/socket.h) was used to evaluate how well we can rebuild a documented API used by the source code. Our tool identified 22 function usage patterns involving only functions defined in the socket API. With these patterns we are able to identify the basic structure of a socket based program.

Wine uses both UDP and TCP communication which causes some functions usage patterns (one involving connect, which is not involved in UDP connections, for example) to have lower confidence values than would be expected in TCP only code.

Figure 26 shows the basic calling pattern for UDP and TCP programs. The recovered API for the socket library is listed in Table 17. Each row contains the functions in the pattern and the confidence values *ConfFirst* and *ConfSecond* for the pattern. Notice that the *ConfSecond* values for patterns involving the function `close` are quite low. This is caused by the double duty done by `close` with respect to the file access API and the socket API. Also, because of the structure of the Wine code, the `send` and `recv` function calls do not form function usage patterns with other functions in the socket API. Aside from the functions to send and receive data, the remainder of the API usage can be recovered from the function usage patterns. The flow of function calls (`socket, bind, listen, accept, close`) can be recovered by inspecting the list. According to the patterns, a call to `socket` should be followed by `bind, listen, accept` or `close`. A call to `bind` should be followed by `listen` or `close`. A call to `listen` should be followed by `accept` or `close`. Finally a call to `accept` should be followed by `close`. The function `setsockopt` can be called at anytime to change the options for the specified socket and this is reflected in the patterns. The difficulty presented by this API is pulling apart the UDP/TCP split and the client/server pattern for TCP.

Function1	Function2	Count	ConfFirst	ConfSecond
accept	close	5	0.38	0.01
accept	setsockopt	2	0.50	0.18
bind	close	5	0.29	0.01
bind	listen	4	0.33	0.21
connect	close	13	0.35	0.01
getsockname	accept	2	0.50	0.50
getsockname	close	8	0.50	0.01
listen	accept	2	0.50	0.50
listen	close	7	0.38	0.01
listen	getsockname	3	0.30	0.30
sendto	recvfrom	2	0.10	0.20
setsockopt	recvfrom	2	0.04	0.20
setsockopt	sendto	2	0.08	0.20
setsockopt	setsockopt	3	0.12	0.12
shutdown	close	5	0.36	0.02
socket	accept	2	0.17	0.50
socket	bind	4	0.08	0.33
socket	close	29	0.47	0.02
socket	connect	9	0.20	0.33
socket	getsockname	4	0.14	0.40
socket	listen	5	0.13	0.36
socketpair	close	11	0.60	0.01

Table 17: Function Usage Patterns, Socket API, Wine

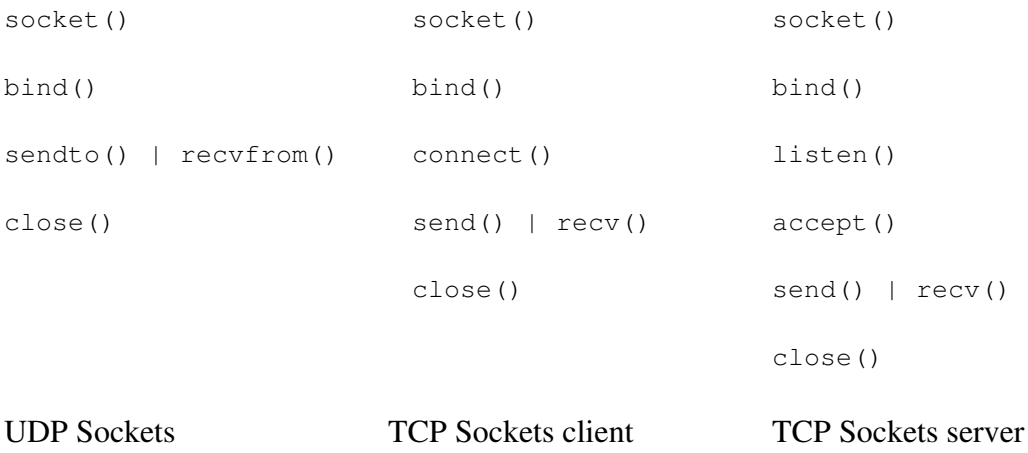


Figure 26: Unix Socket API Usage

5.6 Refactoring

A refactoring is useful to identify and document as it is yet another example of an implicit rule being created to describe the source code. Being able to spot a refactoring as it happens and document this fact to developers who may not interact with the refactored code often can be of great benefit. Further, these new function usage patterns can be used to look for instances where the refactoring was not correctly applied.

Refactoring the source code may consist of adding or removing functions or rewriting a function to alter its functionality. This can cause new function usage patterns to emerge and old patterns to become invalid. We can identify instances of refactoring by examining the instances of function usage patterns mined from the source code history over time. An introduction of a never before seen pattern followed by a number of similar additions may indicate some type of refactoring. Note that either or both functions involved in the refactoring could be newly added to the code. It is also possible for neither function to have been added, as either of the functions could have been significantly rewritten.

We have built a tool to look through the source code repository and identify refactorings. Our tool looks for new instances of never before seen function usage patterns entering the code in large numbers (three or more) very quickly (over a period of one day). The instances of the function usage patterns that we investigate for this must exhibit some type of data flow relationship. Our tool also considers the CVS transaction in which this function usage pattern was first added to determine if either of the functions involved in the pattern are added to the source code in that

transaction. This change profile can indicate the emergence of a new function usage pattern. Once a refactoring is identified, it is possible to determine, from the change history, where bugs created by the refactoring were fixed. These bugs include not updating all the code correctly during the initial refactoring and not fully applying the new function usage pattern later. The fixed bugs identified by this method would all have been identified by the tool we developed in Section 5.4.

5.6.1 Results

Student Projects

We used our tool outlined above to search for refactorings in the student source code repository. In the source code, our tool flagged 128 refactoring events spread across 44 CVS transactions. After examining the source code changes for these CVS transactions, we found that these do in fact highlight the emergence of new function usage patterns that the students need to be aware of. The student CVS transactions do not contain commit messages as these commits were collected automatically by the Marmoset project.

We also identified 17 occurrences where a new instance of one of these function usage patterns identified as part of a refactoring was added to the code after the refactoring occurred. These 17 occurrences were restricted to instances where exactly one function call was added to the source code to create the instance (the manifestation of a bug fix). Each of these 17 instances would have produced a warning with our bug finding tool. Of these 17 new instances, 12 were the result of not applying the refactoring to an existing instance of the code at the time of the refactoring. The remaining 5 new instances were the result of a developer adding half

of the pattern to the source code and subsequently adding the other half of the pattern in a separate CVS transaction.

Apache httpd

We used our tool outlined above to search for refactorings in the Apache httpd source code repository. To determine if we correctly identified a refactoring, we examined the commit message associated with the change as well as inspecting the changed source code.

In the Apache source code, our tool flagged 202 refactoring events. These refactoring events were spread across 45 CVS transactions. After examining both the commit messages and the source code changes for these CVS transactions, we found that these do in fact highlight the emergence of new function usage patterns that the developers need to be aware of. The commit messages for these CVS transactions prove an interesting study. Of the 45 commit messages, 23 contain key words they would lead one to believe that a refactoring of some type had taken place. These keywords are some variation on: rename, introduce, move and add. However, these commit messages are not very specific. Six of the commit messages mention at least one of the functions involved in the new function usage pattern; only one commit message mentions both of the functions. A developer who would like to understand the specific changes made to the source code, and what changes need to be made to her coding habits, would still need to examine the changed code. With our tool in place to analyze each CVS transaction, we can quickly determine that a refactoring has taken place and the specific coding patterns that need to be changed or adopted in response.

We identified 51 occurrences where a new instance of one of the function usage patterns identified as part of a refactoring was added to the code after the refactoring occurred. These 51 occurrences were restricted to instances where exactly one function call was added to the source code to create the instance (the manifestation of a bug fix). Each of these 51 instances would have produced a warning with our bug finding tool. Of these 51 new instances, 32 were the result of not applying the refactoring to an existing instance of the code when the initial refactoring was performed. The remaining 19 new instances were the result of a developer adding half of the pattern to the source code after the refactoring and subsequently adding the other half of the pattern in a separate CVS transaction.

Wine

We used our tool outlined above to search for refactorings in the Wine source code repository. To determine if we correctly identified a refactoring, we examined the commit message associated with the change as well as inspecting the changed source code.

In the Wine source code, our tool flagged 1,342 refactoring events. These refactoring events were spread across 371 CVS transactions. After examining both the commit messages and the source code changes for these CVS transactions, we found that these do in fact highlight the emergence of new function usage patterns that the developers need to be aware of. As with the Apache httpd code, the commit messages for these CVS transactions prove an interesting study. Of the 371 commit messages, 155 contain key words that would lead one to believe that that refactoring of some type had taken place. These keywords are some variation on: rename,

introduce, move and add. However, these commit messages are not very specific. Twenty-two of the commit messages mention at least one of the functions involved in the new function usage pattern; only two commit messages mention both of the functions.

We also identified 242 occurrences where a new instance of one of these function usage patterns identified as part of a refactoring was added to the code after the refactoring occurred. These 242 occurrences were restricted to instances where exactly one function call was added to the source code to create the instance (the manifestation of a bug fix). Each of these 242 instances would have produced a warning with our bug finding tool. Of these 242 new instances, 12 were the result of not applying the refactoring to an existing instance of the code at the time of the refactoring. The remaining 230 new instances were the result of a developer adding half of the pattern to the source code and subsequently adding the other half of the pattern in a separate CVS transaction.

5.7 Threats to Validity

This section discusses a number of threats to the validity of our experiments. The warning messages produced by our tool were inspected by the author, who is not an expert on the code base being reviewed. 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.

Our line number mapping algorithm is based on the normalized minimum edit distance between two source lines and is not perfect. A renamed variable with an especially long name (one greater than 30% of the total length of the line) may cause our tool to mismatch lines or not match lines at all. We have also not tried to identify instances of function renaming. Since we are doing our line number matching on a per-function basis, if a function is renamed it will appear to our tool as a completely new function with no line number mapping back to the previous version of the file. This will cause all the instances of patterns in that function to appear as new instances, albeit instances created by adding both function calls at the same time.

The threshold values used to select the function usage patterns have been determined experimentally by examining the results produced by a much larger selection of function usage patterns, however a full sensitivity analysis was not performed. The thresholds were chosen to reflect the area in the results that provided the most benefit to the developer. In effect, these thresholds act as knobs used to tune the performance of the tools. These threshold values may differ between software projects. However, we believe the ones used here are a good starting point for analysis. Each of the case studies in this work used the same threshold values. The values were not tuned to each software project separately.

The original design of the work called for the parameters to be tuned on the Apache httpd code base and then applied to the Wine code base to determine their correctness. This did not occur as planned. The development of the function usage pattern miner itself became an iterative process, applying the tool to both projects and then refining the tool. This refinement included adding a data flow component and

removing a distance metric from the pattern miner. Because of this, while the parameters were being tuned on Apache httpd we already had substantial knowledge of the Wine software project.

5.8 Computation Costs

A general overview of the computation costs of mining these software repositories are discussed in Section 3.4. Mining the entire source code repository for instances of function usage patterns and determining which instances of patterns is new is a relatively expensive operation. Mining any single version of a file and determining the new instances takes a few tens of seconds. Calculating the confidence statistics for the mined patterns took around 12 hours (using 10 nodes in the cluster) for each of the open source projects.

Analyzing the latest version of the source code broken instances of a single function usage pattern takes a few seconds. This analysis is done by running a series of database queries over the function usage pattern data mined from the latest version of the source code. Searching for violations of large numbers of function usage patterns can be expensive. Sorting the warnings produced by this analysis using the historical information takes a trivial amount of time. This consists of a small constant number of database queries and a sort operation.

Determining the function usage patterns defined for functions implemented in a particular API can be achieved with a single database query and sort operation. For each of the open source operations this query ran for less than an hour. This query was run on a single node in the cluster.

Identifying a refactoring event requires a series of database queries which took around three hours, running on a single node, for each of the open source projects. Identifying the missed refactorings and subsequent bug fixes took another series of database queries and around and hour for each of the open source projects.

The student data was quite small and took around two hours to completely analyze using two nodes on the cluster. The depth of the repository and the size of the file, in terms of number of function call sites it contains, are the biggest factors in the runtime of the tools described in this chapter.

5.9 Summary

In this chapter we presented a technique by which developers can mine software repositories to discover common coding idioms that describe how the source code should be used. We put this technique to use in three applications. First, we used the function usage patterns mined from the source code repository to identify bugs in the latest version of the source code. Our evaluation has shown that these mined properties can be used to identify bugs with a reasonable false positive rate. Further, for the Apache and Wine case studies we have shown that, for the top ranked warnings, the false positive rate of a ranking produced with historical information is better than the false positive rate of a ranking that does not use historical information. The difference was statistically significant in both cases. The common causes of false positives and the irregularity of the APIs in the source code were also discussed.

We then used the data produced by mining the software repository to document the internal and external APIs used by the source code. Finally, we analyzed the software repository to identify instances where new function usage patterns emerged and

studied how this lead to new bugs being introduced into the code. The purpose of the latter two applications was to show that analyzing the changes in the source code history can be used to document the state of the code to the developer. This can be a very important task, especially to large, geographically distributed software projects such as Wine and Apache httpd.

The refactoring events identified by our tool show a number of interesting things. First, while the commit messages in the revision histories of the open source projects where not very specific in the changes made to the code, many of them did convey the idea that these commits introduced new idioms into the source code. This gives us confidence that our tool was identifying interesting commits that make a significant change to the source code. Further, the bug fixes found later in the history based on these new patterns shows that these types of events introduce bugs into the code, by not applying the refactoring everywhere or producing new coding idioms that not all developers are aware of. These bugs provide examples of problems our bug finding tool would have helped to find.

The historical information used by the tools in this chapter is a cumulative analysis of the entire history expressed as a series of confidence measures. However, trends in the recent history may provide additional benefit. The refactoring identifier tool takes advantage of trends to a small degree. An increase of bug fixes applied to a particular function usage pattern in a short amount of time may indicate that the pattern has taken on more significance due to other changes in the code and that warnings produced by this pattern should be more highly ranked. This may also just reflect a change in the awareness by the developers of this function usage pattern and the

remaining warnings are no more likely to be bugs. In either case, this invites further study.

The results of running our bug finding tool over the data gathered from the student projects are better than our results with either the Wine or Apache project (12 of 36 warnings were likely bugs). We propose a number of reasons for this. The types of bugs we are looking for are generally quite severe. The use of data flow information to build the function usage patterns suggests that most of our patterns will involve some use of transformation of data, which when missed may be easy to spot for seasoned developers. Or, put more plainly, they are easier to spot for developers that regularly test their code before making a CVS commit, a characteristic we cannot ascribe to these students. Also, the students are working against an artificial deadline: the final submission date for the project. When the deadline arrives it is better for a student to submit anything rather than nothing. Broken code that does something correctly will get them some points and hence it is to their benefit to submit it. This is not the case with the open source developers. Committing partially broken code immediately before a deadline (say, a release) is not to their benefit.

In this chapter, we have shown that using data mined from the source code repository to rank warning produced by a static analysis tool provides a benefit over just examining the latest version of the source code to rank the warnings. The top of the list, the part most likely to be used by the developer, has a higher precision when the warnings are ranked using this historical data. The differences between the historical and non-historical rankings were also found to be statistically significant.

Future Work

The application of the data in this chapter relied on a number of experimentally derived threshold values. These were used to filter the function usage patterns to identify the patterns that are most likely to be valid and useful to the developers. A formal sensitivity analysis needs to be applied to these values to determine if they are set correctly. As part of this, the values need to be trained on one set of software repositories and applied to another set. The full set of threshold values used is listed in Table 18. The table is broken into three sections, one for each tool described in this chapter. Each section contains a list of parameters and values applied to select function usage patterns used by that tool.

Bug Finding	
Parameter	Value
Count	>= 3
DataFlowConfidence	>=0.80
Number of Violations	<= 4
API Documentation	
Parameter	Value
DataFlowConfidence	>=0.80
Refactoring	
Parameter	Value
Count	>=3
DataFlowConfidence	>=0.80

Table 18: Threshold Values

Many of the false positives reported by our bug finding tool could have been avoided with an interprocedural analysis. Adding this capability to our bug finding tool is a necessary step to make this a more useable system. This will require that data flow analysis is done over function call boundaries to determine when data used by a function call is passed through to another function call to be used by a third,

where a function usage pattern has been defined between the first and third function calls.

Handling more elaborate patterns, where a function needs to be followed by any one function defined in a set, will further reduce the number of false positives produced by our bug finding tool and may produce stronger documentation for a particular API. This will make both the bug finding and documentation portion of our results better.

Chapter 6: Conclusions

In this dissertation, we introduced several techniques to mine source code revision histories to identify important properties that have been added to the source code. We have shown that these properties can be used by the developer to improve the software development process. Specifically, we have used these properties to identify bugs in the source code and to provide documentation of the source code.

To evaluate the effectiveness of these techniques, we have applied our tools to two large open source software projects and a set of student projects from the University of Maryland. The open source projects are projects run by a geographically distributed community of developers. Communication between developers on such projects is extremely important, but there appears to be very little in the way of internal documentation describing how the source code works. The student projects are small, well-defined projects with a life span of a few weeks.

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 types 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.

In Chapter 4, we have shown how data mined from a source code repository can improve static analysis tools. With our static checker we have been able to identify likely bugs in the Apache web server and 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. Further, the false positive rate near the top of the list of ranked warnings, where the developer will first look, was lower for the HistoryAware ranking than the naïve ranking.

In Chapter 5, we have shown how source code change histories can be mined to identify meaningful properties that describe the source code, specifically function usage patterns. In this work we have looked at patterns of two function calls, but we have described how larger patterns may be build up from these, allowing tools to look for more complex relationships. A number of meaningful function usage patterns have been identified and used to statically detect bugs in source code. By looking at the revisions history rather than the final version of the source code we can determine how a function usage pattern emerged. Tracking how each instance of a function usage pattern is created, by adding both function calls together or one at a time, can help identify the patterns that cause trouble for developers and may be more likely to be involved in bugs. We have also used these function usage patterns to document the rules for using functions defined in an API, with the goal of preventing bugs

before they are entered into the code. Internal and external APIs have been derived from the change history of the project. Building an API for an external library from the change history has the shortcoming of only being able to give meaningful results for the parts of the API used by the software project. In the future, mining multiple software projects that use the same external API may give a better picture of the full use of the API. Examining how function usage patterns are created over time allowed our tool to identify the emergence of new function usage patterns that the developer will need to be aware of. Our study of the change history revealed cases where our static bug finding tool would have alerted the developers to instances in the code where the patterns were not applied during a refactoring and to instances later in the life of the source code where the function usage pattern was not fully implemented.

The work described here has all been done on historical data. A system that analyzes each source code change as it is made and keeps an up-to-date database of information could be used to provide quick feedback to the developers on the state of the code. By looking at the historical information, we have identified instances, especially with the function usage pattern miner, where such a tool would be able to alert developers to code that missed during a refactoring or where new function usage patterns have been violated.

If historical data mined from the source code repository was applied to a static analysis tool fewer bug fixes may show up in the repository. This would give our tool less data to work with and possibly degrading the results. Collecting data at a more fine-grain level, on compiles or file saves, may continue to capture enough data to be useful to the static analysis tools. However, the work in this dissertation was

done only to show that using this historical information was possible and useful, not to define the exact use case for this data in the future.

Contributions

In this dissertation we have shown that properties that describe the source code can be discovered by mining the change history stored in the source code repository. We demonstrated this by mining two properties, how to handle the return value of a function and function usage patterns, from the repository. The results of mining these properties were applied to two bug finding tools to improve their precision. A key contribution of this work is that we have shown that benefit is added by mining the entire repository as opposed to mining only the latest version of the source code. The differences in precision between the results that used historical information and those that did not were found to be statistically significant. The function usage patterns were also used to document how to use APIs as well as to identify refactorings events in the code. After identifying refactoring events, we have shown that our bug finding tool, had it been in use during the development process, would have identified a number of violations that were subsequently by the developers.

Appendix A

Apache httpd Function Usage Pattern Warnings and Likely Bugs

Function1	Function2	F1LineNo	F2LineNo	OneAdded-Conf	ConfOne or ConfTwo	Percent Broken	Directory	Filename	Likely Bug
ldap_set_option	ldap_unbind_s	0	222	1	1.00	0.33	modules/experimental	util_ldap.c	0
ldap_init	ldap_unbind_s	0	222	1	1.00	0.33	modules/experimental	util_ldap.c	0
apr_filepath_root	apr_stat	274	0	1	1.00	0.40	server	util.c	0
apr_filepath_root	apr_stat	266	0	1	1.00	0.40	modules/dav/fs	repos.c	0
ap_sub_req_lookup_uri	do_rewritelog	201	0	1	0.50	0.25	modules/mappers	mod_dir.c	0
apr_sockaddr_info_get	apr_socket_connect	0	1208	1	0.50	0.29	modules/proxy	proxy_util.c	0
do_expand	do_rewritelog	2128	0	1	0.50	0.38	modules/mappers	mod_rewrite.c	1
do_expand	do_rewritelog	2131	0	1	0.50	0.38	modules/mappers	mod_rewrite.c	1
do_expand	do_rewritelog	2296	0	1	0.50	0.38	modules/mappers	mod_rewrite.c	1
apr_procattr_io_set	apr_procattr_child_errfn_set	0	750	1	0.46	0.17	server	log.c	0
ap_parse_uri	ap_process_request_internal	656	0	1	0.40	0.25	server	protocol.c	0
ap_run_quick_handler	ap_invoke_handler	1650	0	1	0.40	0.33	server	request.c	0
ap_run_quick_handler	ap_process_request_internal	1896	0	1	0.40	0.33	server	request.c	0
dav_auto_checkout	dav_log_err	2276	0	1	0.38	0.14	modules/dav/main	mod_dav.c	1
apr_procattr_io_set	apr_procattr_error_check_set	0	752	1	0.38	0.25	server	log.c	0
strcasecmp	ap_lookup_provider	0	1387	1	0.38	0.25	modules/aaa	mod_auth_digest.c	0
apr_procattr_create	apr_pool_userdata_set	0	852	1	0.38	0.50	modules/generators	mod_cgid.c	0
apr_procattr_create	apr_pool_userdata_set	0	144	1	0.38	0.50	modules/generators	mod_suexec.c	0
apr_socket_create	ap_run_create_connection	0	374	1	0.36	0.33	modules/proxy	proxy_http.c	0
ap_ssi_get_tag_and_value	strcmp	2115	0	1	0.32	0.25	modules/filters	mod_include.c	1
ap_ssi_get_tag_and_value	strcmp	2028	0	1	0.32	0.25	modules/filters	mod_include.c	1
dav_auto_checkout	dav_created	1220	0	1	0.31	0.29	modules/dav/main	mod_dav.c	0
dav_auto_checkout	dav_created	2276	0	1	0.31	0.29	modules/dav/main	mod_dav.c	0
apr_socket_opt_set	apr_socket_close	320	0	1	0.28	0.09	server	mpm_common.c	0

Function1	Function2	F1LineNo	F2LineNo	OneAdded-Conf	ConfOne or ConfTwo	Percent Broken	Directory	Filename	Likely Bug
ldap_set_option	ldap_unbind_s	1226	0	1	0.25	0.33	modules/experimental	util_ldap.c	1
ap_run_quick_handler	ap_invoke_handler	0	529	1	0.17	0.50	modules/http	http_request.c	1
ap_run_quick_handler	ap_invoke_handler	0	500	1	0.17	0.50	modules/http	http_request.c	1
apr_socket_opt_set	ap_log_perror	320	0	1	0.14	0.33	server	mpm_common.c	0
apr_socket_opt_set	ap_log_perror	183	0	1	0.14	0.33	server	connection.c	1
strstr	apr_psprintf	1010	0	1	0.11	0.50	modules/proxy	proxy_util.c	0
strstr	apr_psprintf	1026	0	1	0.11	0.50	modules/proxy	proxy_util.c	0
apr_table_mergen	apr_bucket_immortal_create	907	0	1	0.10	0.33	modules/proxy	proxy_http.c	0
apr_pool_clear	ap_run_post_config	4180	0	1	0.06	0.33	server	core.c	0
apr_socket_opt_set	apr_socket_connect	1425	0	1	0.00	0.17	modules/proxy	proxy_ftp.c	1
apr_socket_create	apr_socket_connect	1416	0	1	0.00	0.22	modules/proxy	proxy_ftp.c	0
apr_socket_create	apr_socket_connect	297	0	1	0.00	0.22	server	listen.c	0
dav_auto_checkout	dav_log_err	0	620	1	0.00	0.33	modules/dav/main	mod_dav.c	0
ap_allow_options	ap_process_request_internal	0	1653	1	0.00	0.33	server	request.c	0
dav_auto_checkin	dav_log_err	0	2084	1	0.00	0.33	modules/dav/main	mod_dav.c	0
dav_auto_checkin	dav_log_err	0	620	1	0.00	0.33	modules/dav/main	mod_dav.c	0
dav_auto_checkin	dav_log_err	0	4074	1	0.00	0.33	modules/dav/main	mod_dav.c	0
dav_auto_checkout	dav_log_err	0	2084	1	0.00	0.33	modules/dav/main	mod_dav.c	0
dav_auto_checkout	dav_log_err	0	4074	1	0.00	0.33	modules/dav/main	mod_dav.c	0
apr_filepath_name_get	(functionpointer)	0	207	1	0.00	0.38	srclib/apr/misc/unix	otherchild.c	0
apr_filepath_name_get	(functionpointer)	0	204	1	0.00	0.38	srclib/apr/misc/unix	otherchild.c	0
apr_filepath_name_get	(functionpointer)	0	212	1	0.00	0.38	srclib/apr/misc/unix	otherchild.c	0
dav_auto_checkout	dav_created	0	4144	1	0.00	0.44	modules/dav/main	mod_dav.c	0
dav_auto_checkout	dav_created	0	3680	1	0.00	0.44	modules/dav/main	mod_dav.c	0
dav_auto_checkout	dav_created	0	4209	1	0.00	0.44	modules/dav/main	mod_dav.c	0
dav_auto_checkout	dav_created	0	3542	1	0.00	0.44	modules/dav/main	mod_dav.c	0

Appendix B

Wine Function Usage Pattern Warnings and Likely Bugs

Function1	Function2	F1LineNo	F2LineNo	OneAdded-Conf	ConfOne or ConfTwo	Percent Broken	Directory	Filename	Likely Bug
SetStyle	CheckToolbar	0	1849	1.00	1.00	0.20	dlls/shell32	shlview.c	1
strarray_addall	spawn	0	533	1.00	1.00	0.25	tools/winegcc	winegcc.c	0
GetSystemMetrics	ICO_ExtractIconExW	0	615	1.00	0.60	0.40	dlls/user	exticon.c	0
GetSystemMetrics	ICO_ExtractIconExW	0	663	1.00	0.60	0.40	dlls/user	exticon.c	0
LISTVIEW_GetItemW	LISTVIEW_GetStringWidthT	0	8930	1.00	0.60	0.50	dlls/comctl32	listview.c	1
LISTVIEW_GetItemW	LISTVIEW_GetStringWidthT	0	8933	1.00	0.60	0.50	dlls/comctl32	listview.c	1
wine_dbg_log	EXCEPTION_ctor	0	1110	1.00	0.55	0.22	dlls/msvcrt	cpp.c	1
wine_dbg_log	EXCEPTION_ctor	0	1104	1.00	0.55	0.22	dlls/msvcrt	cpp.c	1
wine_dbg_log	mciGetDriverData	0	223	1.00	0.50	0.20	dlls/winmm/mciavi	mciavi.c	1
wine_dbg_log	_dump_DIDATAFORMAT	0	252	1.00	0.50	0.20	dlls/dinput	joystick_linux.c	1
MONTHCAL_HitTest	InvalidateRect	1990	0	1.00	0.50	0.25	dlls/comctl32	monthcal.c	0
debugstr_sockaddr	ws_sockaddr_ws2u	981	0	1.00	0.50	0.25	dlls/winsock	socket.c	1
wine_dbg_log	MENU_MenuBarCalcSize	0	1443	1.00	0.50	0.33	dlls/user	menu.c	1
MONTHCAL_CalcDayRect	MONTHCAL_DrawDay	263	0	1.00	0.50	0.43	dlls/comctl32	monthcal.c	0
MONTHCAL_CalcDayRect	MONTHCAL_DrawDay	665	0	1.00	0.50	0.43	dlls/comctl32	monthcal.c	1
MONTHCAL_CalcDayRect	MONTHCAL_DrawDay	338	0	1.00	0.50	0.43	dlls/comctl32	monthcal.c	1
GetItemPath	RefreshListView	264	0	1.00	0.50	0.50	programs/regedit	treeview.c	1
GetItemPath	RefreshListView	271	0	1.00	0.50	0.50	programs/regedit	childwnd.c	1
MCIAVI_mciStop	MCIAVI_mciGetOpenDev	207	0	1.00	0.46	0.11	dlls/winmm/mciavi	mciavi.c	0
MCIAVI_mciStop	MCIAVI_mciGetOpenDev	379	0	1.00	0.46	0.11	dlls/winmm/mciavi	mciavi.c	0
MCIAVI_mciStop	MCIAVI_mciGetOpenDev	284	0	1.00	0.46	0.11	dlls/winmm/mciavi	mciavi.c	0
EDIT_BuildLineDefs_ML	EDIT_UpdateText	2208	0	1.00	0.45	0.14	dlls/user	edit.c	1
wine_dbg_log	CallWindowProcA	0	1089	0.95	0.85	0.04	dlls/commdlg	fontdlg.c	1

Function1	Function2	F1LineNo	F2LineNo	OneAdded-Conf	ConfOne or ConfTwo	Percent Broken	Directory	Filename	Likely Bug
wine_dbg_log	CallWindowProcA	0	1102	0.95	0.85	0.04	dlls/commdlg	fontdlg.c	1
wine_dbg_log	CallWindowProcA	0	1203	0.95	0.85	0.04	dlls/commdlg	colordlg.c	1
GetPropW	PROPSHEET_SetCurSel	0	3395	0.83	1.0	0.33	dlls/comctl32	propsheet.c	0
GetPropW	PROPSHEET_SetCurSel	0	3337	0.83	1.0	0.33	dlls/comctl32	propsheet.c	0
GetWindowLongA	PAGER_SetPos	0	667	0.83	1.0	0.50	dlls/comctl32	pager.c	0
GetWindowLongA	PAGER_SetPos	0	1521	0.83	1.0	0.50	dlls/comctl32	pager.c	0
wine_dbg_log	HEAP_Dump	0	272	0.83	0.75	0.20	dlls/ntdll	heap.c	1
TOOLTIPS_GetToolFromInfoW	wine_dbg_log	1170	0	0.83	0.50	0.29	dlls/comctl32	tooltips.c	0
TOOLTIPS_GetToolFromInfoW	wine_dbg_log	1434	0	0.83	0.50	0.29	dlls/comctl32	tooltips.c	0
GetWindowLongA	TAB_EnsureSelectionVisible	0	469	0.83	0.50	0.40	dlls/comctl32	tab.c	0
GetWindowLongA	TAB_EnsureSelectionVisible	0	237	0.83	0.50	0.40	dlls/comctl32	tab.c	0
wine_dbg_log	MSSTYLES_FindProperty	0	148	0.80	0.54	0.16	dlls/uxtheme	draw.c	1
wine_dbg_log	MSSTYLES_FindProperty	0	131	0.80	0.54	0.16	dlls/uxtheme	draw.c	1
wine_dbg_log	MSSTYLES_FindProperty	0	162	0.80	0.54	0.16	dlls/uxtheme	draw.c	1
new_res	set_dword	197	0	0.80	0.50	0.09	tools/wrc	readres.c	1
new_res	get_dword	62	0	0.80	0.50	0.09	tools/wrc	writeres.c	0
new_res	get_dword	197	0	0.80	0.50	0.09	tools/wrc	readres.c	0
new_res	set_dword	62	0	0.80	0.50	0.09	tools/wrc	writeres.c	1
new_res	set_dword	0	497	0.80	0.48	0.13	tools/wrc	genres.c	0
put_res_header	set_dword	0	497	0.80	0.48	0.13	tools/wrc	genres.c	0
put_res_header	set_dword	0	478	0.80	0.48	0.13	tools/wrc	genres.c	1
new_res	set_dword	0	479	0.80	0.48	0.13	tools/wrc	genres.c	0
new_res	set_dword	0	478	0.80	0.48	0.13	tools/wrc	genres.c	0
put_res_header	set_dword	0	479	0.80	0.48	0.13	tools/wrc	genres.c	1
MF_GetMetaHeader	wine_dbg_log	617	0	0.75	1.00	0.25	dlls/gdi	metafile.c	1
GlobalFindAtomA	debugstr_a	1659	0	0.75	0.50	0.33	windows	win.c	0
GlobalFindAtomA	debugstr_a	724	0	0.75	0.50	0.33	windows	class.c	1

Bibliography

[1] Agrawal, R., Imielinski, T., Swami, A., Mining Association Rules between Sets of Items in Large Databases (1993). *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, May 26–28 1993, pp.207–216.

[2] Agrawal, R., Srikant, R., Fast Algorithms for Mining Association Rules, Proceeding of the 20th Very Large Database Conference (VLDB '94), Santiago, Chile, 1994.

[3] Ammons, G., Bodik, R., Larus, J. R., Mining Specifications, *Proceedings of Principles of Programming Languages* (POPL '02). Portland, OR, USA. Jan 2002.

[4] Apache Portable Runtime, APR. Available online at http://apr.apache.org

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

[6] 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.

[7] Ball, T., The Concept of Dynamic Analysis, Proceedings of Foundations of Software Engineering (FSE '99), Toulouse, France, Sept 1999. p 216-234.

[8] Ball, T., Kim, J., Porter, A., Siy, H.,. If your version control system could talk. In *Proceedings Workshop on Process Modeling and Empirical Studies of Software Engineering*, May 1997.

[9] 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.

[10] 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.

[11] Buck, B., Hollingsworth, J.K., ``An API for Runtime Code Patching," *http://www.dyninst.org/papers/apiPreprint.pdf*

[12] Bush, W. R., Pincus, J. D., Sielaff, D. J., A static analyzer for finding dynamic programming errors, *Software Practice and Experience*, June 2000, vol. 30, number 7, pages 775-802.

[13] Bugzilla, Available online at http://www.bugzilla.org/

[14] Brun, Y., Ernst, M. D., Finding latent code errors via machine learning over program executions, *Proceedings of International Conference of Software Engineering* (ICSE '04), Edinburgh, Scotland, UK, May 2004.

[15] Chen, A., Chou, E. Wong, J., Yao, A. Y., Zhang, Q., Zhang, S., Michal, A., CVSSearch: Searching through Source Code using CVS Comments, In Proceedings IEEE International Conference on Software Maintenance (ICSM'01), November 7 - 9, 2001, Florence, Italy , pages 364 – 373.

[16] Crew, R. F., ASTLOG: A Language for Examining Abstract Syntax Trees, *Proceedings of USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[17] Cubranic, D., Project History as a Group Memory: Learning From the Past, PhD thesis, University of British Columbia, 2004.

[18] CVS – Concurrent Versions System. Available online at http://www.cvshome.org

[19] CVSSearch II: A Search Engine for Code. Available online at http://cvssearch.sourceforge.net/

[20] Descartes, A., Bunce, T. Programming the Perl DBI, O'Reilly, 2000.

[21] Detlef, D. L., An overview of the Extended Static Checking System, *Proceedings of Workshop on Formal Methods in Software Practice*, Jan 1996. pages 1-9.

[22] Edison Design Group, http://www.edg.com/cpp.html

[23] 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.

[24] Engler, D., Chen, D. Y., Hallem, S., Chou, A., Chelf, B., Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, In Proceedings of the ACM symposium on Operating Systems Principles, Banff, Canada, Oct 2001.

[25] Ernst, M. D., Static and dynamic analysis: synergy and duality, Invited talk, In *Proceedings Workshop on Program Analysis for Software Tools and Engineering,* Washington, DC, June 2004.

[26] Ernst, M. D., Cockrell, J., Griswold, W. G., Notkin, D., Dynamically Discovering Likely Program Invariants to Support Program Evolution, *IEEE Transactions on Software Engineering*, vol. 27 number 2, Feb 2001.

[27] Ferenc, R., Siket, I., Gyimothy, T., Extracting Facts from Open Source Software, In Proceedings of 20th International Conference on Software Maintenance (ICSM'04), Sept 2004, Chicago, Illinois, USA, pages 60-69.

[28] Fischer, M., Pinzger, M., Gall, H., Populating a Release History Database from Version Control and Bug Tracking Systems, *Proceedings of International Conference on Software Maintenance* (ICSM '03), Amsterdam, The Netherlands, Sept 2003.

[29] Fischer, M., Gall, H., Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data. Journal of Software Maintenance and Evolution: Research and Practice, 16:385-403, John Wiley & Sons, Ltd., November/December 2004.

[30] Fischer, M., Pinzger, M., Gall, H., Analyzing and Relating Bug Report Data for Feature Tracking, In Proceedings of 10th Working Conference on Reverse Engineering (WCRE'03), Nov 2003, Victoria, B.C., Canada, pages 90-99.

[31] 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.

[32] German, D. M., An Empirical Study of Fine-Grained Software Modifications, In Proceedings of 20th International Conference on Software Maintenance (ICSM'04), Sept 2004, Chicago, Illinois, USA, pages 316-325

[33] Godfrey, M., Dong, X., Kapser, C., Zou, L., Four Interesting Ways in Which History Can Teach Us About Software, *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[34] 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

[35] Gutwin, C., Penner, R., Schneider, K., Group Awareness in Distributed Software Development, In Proceedings of ACM Conference on Computer Supported Cooperative Work, Chicago, IL, Nov 2004.

[36] Hangal, S., Lam, M. S., Tracking Down Software Bugs Using Automatic Anomaly Detection*, Proceedings International Conference on Software Engineering* (ICSE '02). Orlando, FL, USA, May 2002.

[37] Hassan, A. E., Holt, R. C., The Top Ten List: Dynamic Fault Prediction, *submitted for publication*, http://plg.uwaterloo.ca/~aeehassa/home/pubs/fase2004.pdf.

[38] Hassan, A.E., Holt, R.C., Predicting Change Propagation in Software Systems, In Proceedings of 20th International Conference on Software Maintenance (ICSM'04), Sept 2004, Chicago, Illinois, USA, pages 284-293

[39] 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.

[40] Holzmann, G. J., Static Source Code Checking For User-Defined Properties, *Proceedings of Integrated Design and Process Technology* (IDPT '02), Pasadena, CA, USA. June 2002.

[41] Hovemeyer, D., Pugh, W., Finding Bugs Is Easy, In Companion of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), Vancouver, British Columbia, Canada, October 2004.

[42] Johnson, S., Unix Time Sharing System Programmer's Manual, AT&T Bell Laboratories, 1979. Seventh Edition, Volume 2A.

[43] Kim, S., Pan, K., Whitehead, Jr., E. J., Micro Pattern Evolution, *Proceedings of International Workshop on Mining Software Repositories* (MSR '06), Beijing, China, May 2006.

[44] 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.

[45] Li, Z., Zhou, Y., PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code, In Proceedings of the *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2005)*, Lisbon, Portugal, September 2005.

[46] Liu, Y., Stroudia, E., Wong, K., German, D., Using CVS Historical Information to Understand How Students Develop Software, *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[47] Livshits, B., Zimmermann, T., DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In Proceedings of the *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2005)*, Lisbon, Portugal, September 2005.

[48] Mandelin, D., Xu, L., Bodik, R., Kimelman, D., Jungloid Mining: Helping to Navigate the API Jungle, In Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05), Chicago, IL, USA, June 12-15, 2005.

[49] Matsumura, T., Monden, A., Matsumoto, K., The Detection of Faulty Code Violating Implicit Coding Rules, Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02), Orlando, FL, USA, May 2002. pages 15-21.

[50] Menzies, T., Ammar, K., Nikora, A. , Di Stefano, J., How Simple is software defect detection? Submitted to *Empirical Software Engineering Journal*. 2003. http:// menzies.us/pdf/03simpled.pdf

[51] Menzies, T., Di Stefano, J. S., Cunanan, C., Chapman, R., Mining Repositories to Assist in Project Planning and Resource Allocation, Proceedings of International Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 2004.

[52] Michail, A., Data Mining Library Reuse Patterns in User-Selected Applications, In Proceedings of the International Conference on Software Engineering, June 2000.

[53] Nimmer, J. W., Ernst, M. D., Automatic Generation of Program Specification, *Proceedings of International Symposium on Software Testing and Analysis* (ISSTA '02), Rome, Italy, July 2002, pages 232-242.

[54] Ohira, M., Yokomori, R., Sakai, M., Matsumoto, K., Inoue, K, Torii, K., Empirical Project Monitor: A Tool for Mining Multiple Projects Data, *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[55] OpenSSL Project, Available online at http://www.openssl.org/

[56] Ostrand, T. J., Weyuker, E. J., A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files, *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[57] Ostrand, T. J., Weyuker, E. J., Bell, R. M., Where the Bugs Are, Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '04), Boston, MA, USA, July 2004.

[58] PBS, Available online at http://www.openpbs.org/

[59] Pinzger, M., Gall, H., Pattern-supported architecture recovery. In Proceedings of the International Workshop on Program Comprehension (IWPC'02), Paris, France, June 2002.

[60] Purushothaman, R., Perry, D. E., Towards Understanding the Rhetoric of Small Changes, Proceedings of International Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 2004.

[61] 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.

[62] RCS, Available online at http://www.cs.purdue.edu/homes/trinkle/RCS/index.html

[63] Rysselberghe, F., Demeyer, S., Mining Version Control Systems for FACs (Frequently Applied Changes), Proceedings of International Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 2004.

[64] Shirabad, J. S., Lethbridge, T. C., Matwin, S., Mining the Software Change Repository of a Legacy Telephony *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[65] Spacco, J., Strecker, J., Hovemeyer, D., Pugh, W. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *Proceedings of the Mining Software Repositories Workshop (MSR 2005)*, St. Louis, Missouri, USA, May 2005.

[66] Stallman, R. M., Using the GNU Compiler Collection, GNU Press, 2004.

[67] Whaley, J., Martin, M. C., Lam, M. S., Automatic Extraction of Object-Oriented Component Interfaces, Proceedings of International Symposium on Software Testing and Analysis, Rome, Italy, July 2002, pages 218-228.

[68] Widenius, M., Axmark, D. MySQL Reference Manual Documentation from the Source, O'Reilly, 2002.

[69] Williams, C. C., Hollingsworth, J. K., Bug Driven Bug Finders, *Proceedings of International Workshop on Mining Software Repositories* (MSR '04), Edinburgh, Scotland, UK, May 2004.

[70] Williams, C. C., Hollingsworth, J. K., Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques, IEEE Transactions on Software Engineering, 31(6), June 2005.

[71] Williams, C. C., Hollingsworth, J. K., Recovering System Specific Rules from Software Repositories, In Proceedings of International Workshop on Mining Software Repositories (MSR '05), St. Louis, MO, May 2005.

[72] Wine, Available online at http://www.winehq.org

[73] Xie, T., Pei, J., MAPO: Mining API Usages from Open Source Repositories, *Proceedings of International Workshop on Mining Software Repositories* (MSR '06), Beijing, China, May 2006.

[74] 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.

[75] Zimmermann, T., Weissgerber, P., Preprocessing CVS Data for Fine-Grained Analysis, Proceedings of International Workshop on Mining Software Repositories (MSR '04), Edinburgh, Scotland, UK, May 2004.

[76] Zimmerman, T., Weissgerber, P., Diehl, S., Zeller, A., Mining Version Histories to Guide Software Changes, *Proceedings of International Conference of Software Engineering* (ICSE '04), Edinburgh, Scotland, UK, May 2004.